



Community Experience Distilled

Boost.Asio C++ Network Programming

Enhance your skills with practical examples for C++ network programming

John Torjo

[PACKT] open source*
PUBLISHING community experience distilled

Boost.Asio C++ Network Programming

Enhance your skills with practical examples
for C++ network programming

John Torjo



BIRMINGHAM - MUMBAI

Boost.Asio C++ Network Programming

Copyright © 2013 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: February 2013

Production Reference: 1120213

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78216-326-8

www.packtpub.com

Cover Image by J.Blaminsky (milak6@wp.pl)

Credits

Author

John Torjo

Project Coordinator

Sherin Padayatty

Reviewers

Béla Tibor Bartha

Nicolae Ghimbovski

Proofreader

Claire Cresswell-Lane

Acquisition Editor

Erol Staveley

Indexer

Monica Ajmera Mehta

Commissioning Editor

Ameya Sawant

Graphics

Valentina D'silva

Aditi Gajjar

Technical Editor

Kaustubh S. Mayekar

Production Coordinator

Conidon Miranda

Cover Work

Conidon Miranda

About the Author

John Torjo is a renown C++ expert. He has been programming for over 15 years, most of which were spent doing C++. Sometimes, he also codes C# or Java.

He's also enjoyed writing articles about programming in C++ Users Journal (currently, Dr. Dobbs) and other magazines.

In his spare time, he likes playing poker and driving fast cars. One of his freelance projects lets him combine two of his passions, programming and poker. You can reach him at `john.code@torjo.com`.

I'd like to thank my friends Alexandru Chis, Aurelian Hale, Bela Tibor Bartha, Cristian Fatu, Horia Uifaleanu, Nicolae Ghimbovski, and Ovidiu Deac for their feedback and suggestions relating to the book. I'd also like to thank the guys at Packt for being understanding, even though I missed a few deadlines now and then. And many thanks to Chris Kohlhoff, the author of Boost.Asio, for writing such a damn good library!

I dedicate the book to my best friend, Darius

About the Reviewers

Béla Tibor Bartha is a professional software engineer working on various technologies and languages. Although, in the last four years, he's working on iOS and OSX applications, as C++ is his old passion along with game development as personal projects.

I would like to thank John for the possibility to review this book.

Nicolae Ghimbovski is a talented individual, who has been working on various C/C++ projects for over 5 years. He has been involved mostly in telecommunication projects for enterprises. He is a dedicated Linux hobbyist, who enjoys testing and experimenting different operating systems, scripting tools, and programming languages. Besides programming, he enjoys cycling, yoga, and meditation.

I would like to thank John for letting me to review his book.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at service@packtpub.com for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<http://PacktLib.PacktPub.com>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface	1
Chapter 1: Getting Started with Boost.Asio	5
What is Boost.Asio?	5
History	6
Dependencies	7
Building Boost.Asio	7
Important macros	8
Synchronous versus asynchronous	8
Exceptions versus error codes	11
Threading in Boost.Asio	12
Not just networking	13
Timers	14
The io_service class	15
Summary	19
Chapter 2: Boost.Asio Fundamentals	21
The Network API	21
Boost.Asio namespaces	21
IP addresses	22
Endpoints	22
Sockets	23
Synchronous error codes	24
Socket member functions	24
Other considerations	31
The read/write/connect free functions	35
The connect functions	35
The read/write functions	36

Asynchronous programming	40
The need for going asynchronous	40
Asynchronous run(), run_one(), poll(), poll_one()	44
Running forever	44
The run_one(), poll(), poll_one() functions	45
Asynchronous work	47
Asynchronous post() versus dispatch() versus wrap()	50
Staying alive	52
Summary	55
Chapter 3: Echo Server/Clients	57
TCP Echo server/clients	58
TCP synchronous client	59
TCP synchronous server	60
TCP asynchronous client	61
TCP asynchronous server	64
The code	66
UDP Echo server/clients	66
UDP synchronous Echo client	67
UDP synchronous Echo server	68
Summary	68
Chapter 4: Client and Server	69
The synchronous client/server	70
Synchronous client	70
Synchronous server	73
The asynchronous client/server	77
Asynchronous client	78
Asynchronous server	82
Summary	86
Chapter 5: Synchronous Versus Asynchronous	87
Mixing synchronous and asynchronous programming	87
Passing client to server messages and vice versa	88
Synchronous I/O in client applications	89
Synchronous I/O in server applications	92
Threading in a synchronous server	94
Asynchronous I/O in client applications	96
Asynchronous I/O in server applications	98
Threading in an asynchronous server	101
Asynchronous operations	104
Implementing proxies	108
Summary	111

Chapter 6: Boost.Asio – Other Features	113
std streams and std buffer I/O	113
Boost.Asio and the STL streams	114
The streambuf class	116
The free functions that deal with streambuf objects	118
Co-routines	120
Summary	125
Chapter 7: Boost.Asio – Advanced Topics	127
Asio versus Boost.Asio	127
Debugging	128
Handler tracking information	128
An example	129
Handler tracking to file	131
SSL	132
Boost.Asio Windows features	133
Stream Handles	134
Random access Handles	134
Object Handles	135
Boost.Asio POSIX features	135
Local sockets	135
Connecting local sockets	136
POSIX file descriptors	136
Fork	137
Summary	138
Index	139

Preface

Network programming has been around for a very long time, and it's definitely not a task for the faint-hearted. Boost.Asio provides an excellent abstraction over it, making sure that with a minimal amount of coding, you can create beautiful client-server applications and have tons of fun doing it. And it throws some extra non-networking features, just as a bonus! Code that uses Boost.Asio is compact, easy to read, and if you follow what I describe in the book, it is bug-free.

What this book covers

Chapter 1, Getting Started with Boost.Asio will present what Boost.Asio is, how to build it, and a few examples along the way. Boost.Asio is more than a networking library as you're about to find out. You'll also discover the most important class that sits at the heart of Boost.Asio, `io_service`.

Chapter 2, Boost.Asio Fundamentals will cover what you definitely need to know in order to know when using Boost.Asio. We'll delve deeper into asynchronous programming, which is trickier than synchronous and is much more fun. This chapter was implemented as a reference, which you should come back to, while implementing your own networking applications.

Chapter 3, Echo Server/Clients will implement you to implement a small client-server application; probably, the easiest client-server application you will ever write. This is the Echo application, which is a server that echoes back anything a client writes and then closes the client's connection. We will implement first a synchronous application, and then an asynchronous application, so you can easily compare them.

Chapter 4, Client and Server will discuss delving into building non-trivial client and server applications using Boost.Asio. We will discuss how to avoid pitfalls, such as memory leaks and deadlocks. All the programs are meant to be skeletons you can extend and adapt to your needs.

Chapter 5, Synchronous Versus Asynchronous will walk you through the things to consider when choosing to go synchronous or asynchronous. First off, avoid mixing them. In this chapter, we'll see how easy it can be to implement, test, and debug each type of application.

Chapter 6, Boost.Asio Other Features will walk you through some of the not-so-well-known features of Boost.Asio. `std` streams and streambufs can be a bit more complicated to use, but as you'll see, they bring their own benefits to the table. Finally, you'll see a rather late entry to Boost.Asio, that is, co-routines, which allow you to have code that is asynchronous, but is much easier to read (as if it was synchronous).

Chapter 7, Boost.Asio Advanced Topics will deal with some of the advanced topics of Boost.Asio. It's unlikely that you'll need to delve into these for day-to-day programming, but they are definitely good to know (advanced debugging Boost.Asio, SSL, Windows-only features, and POSIX-only features).

What you need for this book

In order to compile Boost.Asio and run the examples that come with this book, you'll need a modern compiler. For instance, Visual Studio 2008+ or g++ 4.4+.

Who this book is for

This book is great for developers that need to do network programming but don't want to delve into the complicated issues of raw networking API. What you want is an easy abstraction, which is just what Boost.Asio provides. Being part of the famous Boost C++ Library, chances are switching to Boost.Asio is just a few extra `#include` directives.

In order to read the book, you should be familiar with the core Boost libraries, such as Boost Smart Pointers, `boost::noncopyable`, Boost Functors, Boost Bind, `shared_from_this/enabled_shared_from_this`, and Boost Threading (threads and mutexes). A bit of familiarity with Boost Date/Time is required as well. Readers should also be familiar with the concept of blocking versus "non-blocking" operations.

Conventions


In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.


Code words in text are shown as follows: "Usually one instance of `io_service` will be enough."

A block of code is set as follows:

```
read(stream, buffer [, extra options])
async_read(stream, buffer [, extra options], handler)
write(stream, buffer [, extra options])
async_write(stream, buffer [, extra options], handler)
```

New terms and **important words** are shown in bold.

[ Warnings or important notes appear in a box like this.]

[ Tips and tricks appear like this.]

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book – what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to feedback@packtpub.com, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books – maybe a mistake in the text or the code – we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/submit-errata>, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from <http://www.packtpub.com/support>.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at copyright@packtpub.com with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at questions@packtpub.com if you are having a problem with any aspect of the book, and we will do our best to address it.

1

Getting Started with Boost.Asio

First, let's delve into what Boost.Asio is, how to build it, and a few examples along the way. Boost.Asio is more than a networking library as you're about to find out. You'll also discover the most important class that sits at the heart of Boost.Asio - `io_service`.

What is Boost.Asio?

In short, Boost.Asio is a cross-platform C++ library mainly for networking and some other low-level input/output programming.

There have been many implementations that have tackled networking, but Boost.Asio has by far surpassed them all; it was admitted into Boost in 2005, and has since been tested extensively by Boost users and used in many projects, such as Remobo (<http://www.remobo.com>) that allows you to create your own **Instant Private Network (IPN)**, **libtorrent** (<http://www.rasterbar.com/products/libtorrent>), which is a library that implements a Bittorrent client, and PokerTH (<http://www.pokerth.net>), which is a poker game that supports LAN and Internet games.

Boost.Asio has successfully abstracted the concepts of input and output that work not just for networking but for COM serial ports, files, and so on. On top of these, you can do input or output programming synchronously or asynchronously:

```
read(stream, buffer [, extra options])
async_read(stream, buffer [, extra options], handler)
write(stream, buffer [, extra options])
async_write(stream, buffer [, extra options], handler)
```




Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

As you can see in the preceding code snippet, the functions take a `stream` instance, which can be anything (not just a socket, we can read or write to it).

The library is portable and works across most operating systems, and scales well over thousands of concurrent connections. The networking part was inspired by **Berkeley Software Distribution (BSD)** sockets. It provides an API that deals with **Transmission Control Protocol (TCP)** sockets, **User Datagram Protocol (UDP)** sockets, **Internet Control Message Protocol (ICMP)** sockets, and is extensible as you can adapt it to your own protocol if you wish.

History

Boost.Asio was accepted into Boost 1.35 in December 2005, after being developed in 2003. The original author is *Christopher M. Kohlhoff*, and he can be reached at chris@kohlhoff.com.

The library has been tested on the following platforms and compilers:

- 32-bit and 64-bit Windows, using Visual C++ 7.1 and above
- Windows using MinGW
- Windows using Cygwin (make sure to define `__USE_232_SOCKETS`)
- Linux based on 2.4 and 2.6 kernels, using g++ 3.3 and above
- Solaris, using g++ 3.3 and above
- MAC OS X 10.4+, using g++ 3.3 and above

It may also work on the platforms, such as AIX 5.3, HP-UX 11i v3, QNX Neutrino 6.3, Solaris using Sun Studio 11+, True64 v5.1, Windows using Borland C++ 5.9.2+ (consult at www.boost.org for more details).

Dependencies

Boost.Asio depends on the following libraries:

- **Boost.System:** This library provides operating system support for Boost libraries (http://www.boost.org/doc/libs/1_51_0/doc/html/boost_system/index.html)
- **Boost.Regex:** This library (optional) is used in case you're using the `read_until()` or `async_read_until()` overloads that take a `boost::regex` parameter
- **Boost.DateTime:** This library (optional) is used if you use Boost.Asio timers
- **OpenSSL:** This library (optional) is used if you decide to use the SSL support provided by Boost.Asio

Building Boost.Asio

Boost.Asio is a header-only library. However, depending on your compiler and the size of your program, you can choose to build in Boost.Asio as a source file. You may want to do this to decrease the compilation times. This can be done in the following ways:

- In only one of your files, by using `#include <boost/asio/impl/src.hpp>` (if you're using SSL, `#include <boost/asio/ssl/impl/src.hpp>` as well)
- By using `#define BOOST_ASIO_SEPARATE_COMPILATION` in all your source files

Do note that Boost.Asio depends on Boost.System and optionally Boost.Regex, so you'll need to at least build boost libraries, using the following code:

```
bjam -with-system -with-regex stage
```

If you want to build the tests as well, you should use the following code:

```
bjam -with-system -with-thread -with-date_time -with-regex -with-serialization stage
```

The library comes with lots of examples, which you can check out, along with the examples that come with this book.

Important macros

Use `BOOST_ASIO_DISABLE_THREADS` if set; it disables threading support in Boost.Asio, regardless of whether Boost was compiled with threading support.

Synchronous versus asynchronous

First off, asynchronous programming is extremely different than synchronous programming. In synchronous programming, you do the operations in sequential order, such as read (request) from socket *S*, then write (answer) to socket. Each operation is blocking. Since operations are blocking, in order not to interrupt the main program while you're reading from or writing to a socket, you'll usually create one or more threads that deal with socket's input/output. Thus, synchronous servers/clients are usually multi-threaded.

In contrast, asynchronous programming is event-driven. You start an operation, but you don't know when it will end; you supply a callback, which the API will call when the operation ends, together with the operation result. To programmers that have extensive experience with QT, Nokia's cross-platform library for creating graphical user interface applications, this is second nature. Thus, in asynchronous programming, you don't necessarily need more than one thread.

You should decide early on in your project (preferably at the start) whether you go synchronous or asynchronous with networking, as switching midway will be very difficult and error-prone; not only will the API differ substantially, the semantic of your program will change completely (asynchronous networking is usually harder to test and debug than synchronous networking). You'll want to think of either going for blocking calls and multi-threading (synchronous, usually simpler) or less-threads and events (asynchronous, usually more complex).

Here's a basic example of a synchronous client:

```
using boost::asio;
io_service service;
ip::tcp::endpoint ep( ip::address::from_string("127.0.0.1"), 2001);
ip::tcp::socket sock(service);
sock.connect(ep);
```

First, your program needs at least an `io_service` instance. Boost.Asio uses `io_service` to talk to the operating system's input/output services. Usually one instance of an `io_service` will be enough. Next, create the address and port you want to connect to. Create the socket. Connect the socket to your address and port:

```

Here is a simple synchronous server:using boost::asio;
typedef boost::shared_ptr<ip::tcp::socket> socket_ptr;
io_service service;
ip::tcp::endpoint ep( ip::tcp::v4(), 2001); // listen on 2001
ip::tcp::acceptor acc(service, ep);
while ( true) {
    socket_ptr sock(new ip::tcp::socket(service));
    acc.accept(*sock);
    boost::thread( boost::bind(client_session, sock));
}

void client_session(socket_ptr sock) {
    while ( true) {
        char data[512];
        size_t len = sock->read_some(buffer(data));
        if ( len > 0)
            write(*sock, buffer("ok", 2));
    }
}

```

Again, first your program needs at least one `io_service` instance. You then specify the port you're listening to, and create an acceptor, one object that accepts client connections.

In the following loop, you create a dummy socket and wait for a client to connect. Once a connection has been established, you create a thread that will deal with that connection.

In the `client_session` thread, read a client's request, interpret it, and answer back.

To create a basic asynchronous client, you'll do something similar to the following:

```

using boost::asio;
io_service service;
ip::tcp::endpoint ep( ip::address::from_string("127.0.0.1"), 2001);
ip::tcp::socket sock(service);
sock.async_connect(ep, connect_handler);
service.run();

void connect_handler(const boost::system::error_code & ec) {
    // here we know we connected successfully
    // if ec indicates success
}

```

Your program needs at least one `io_service` instance. You specify where you connect to and create the socket.

You then connect asynchronously to the address and port once the connection is complete (its completion handler), that is, `connect_handler` is called.

When `connect_handler` is called, check for the error code (`ec`), and if successful, you can write asynchronously to the server.

Note that the `service.run()` loop will run as long as there are asynchronous operations pending. In the preceding example, there's only one such operation, that is, the socket `async_connect`. After that, `service.run()` exits.

Each asynchronous operation has a completion handler, a function that is called when the operation has completed.

The following code is of a basic asynchronous server:

```
using boost::asio;
typedef boost::shared_ptr<ip::tcp::socket> socket_ptr;
io_service service;
ip::tcp::endpoint ep( ip::tcp::v4(), 2001); // listen on 2001
ip::tcp::acceptor acc(service, ep);
socket_ptr sock(new ip::tcp::socket(service));
start_accept(sock);
service.run();

void start_accept(socket_ptr sock) {
    acc.async_accept(*sock, boost::bind( handle_accept, sock, _1 ));
}

void handle_accept(socket_ptr sock, const boost::system::error_code &
err) {
    if ( err) return;
    // at this point, you can read/write to the socket

    socket_ptr sock(new ip::tcp::socket(service));
    start_accept(sock);
}
```

In the preceding code snippet, first, you create an `io_service` instance. You then specify what port you're listening to. Then, you create the `acc` acceptor, an object to accept client connections and also, create a dummy socket, and asynchronously wait for a client to connect.

Finally, run the asynchronous `service.run()` loop. When a client connects, `handle_accept` is called (the completion handler for the `async_accept` call). If there's no error, you can use this socket for read/write operations.

After using the socket, you create a new socket, and call `start_accept()` again, which appends another "wait for client to connect" asynchronous operation, keeping the `service.run()` loop busy.

Exceptions versus error codes

Boost.Asio allows for both exceptions or error codes. All the synchronous functions have overloads that either throw in case of error or can return an error code. In case the function throws, it will always throw a `boost::system::system_error` error.

```
using boost::asio;
ip::tcp::endpoint ep;
ip::tcp::socket sock(service);
sock.connect(ep); // Line 1
boost::system::error_code err;
sock.connect(ep, err); // Line 2
```

In the preceding code, `sock.connect(ep)` will throw in case of an error, and `sock.connect(ep, err)` will return an error code.

Take a look at the following code snippet:

```
try {
    sock.connect(ep);
} catch(boost::system::system_error e) {
    std::cout << e.code() << std::endl;
}
```

The following code snippet is similar to the preceding one:

```
boost::system::error_code err;
sock.connect(ep, err);
if (err)
    std::cout << err << std::endl;
```

In case you're using asynchronous functions, they all return an error code, which you can examine in your callback. Asynchronous functions never throw an exception, as it would make no sense to do so. And who would catch it?

In your synchronous functions, you can use exceptions or error codes (whatever you wish), but do it consistently. Mixing them up can cause problems and most of the time crashes (when you forget to handle a thrown exception, by mistake). If your code is complex (socket read/write function calls), you should probably prefer exceptions and embody your reads/writes in the `try { } catch` block of a function.

```
void client_session(socket_ptr sock) {
    try {
        ...
    } catch ( boost::system::system_error e) {
        // handle the error
    }
}
```

If using error codes, you can very nicely see when the connection is closed, as shown in the following code snippet:

```
char data[512];
boost::system::error_code error;
size_t length = sock.read_some(buffer(data), error);
if (error == error::eof)
    return; // Connection closed
```

All Boost.Asio error codes are in namespace `boost::asio::error` (in case you want to create a big switch to check out the cause of the error). Just check out the `boost/asio/error.hpp` header for more details.

Threading in Boost.Asio

When it comes to threading in Boost.Asio, we will talk about:

- **io_service:** The `io_service` class is thread-safe. Several threads can call `io_service::run()`. Most of the time you'll probably call `io_service::run()` from a single thread that function is blocking until all asynchronous operations complete. However, you can call `io_service::run()` from several threads. This will block all threads that have called `io_service::run()`. All callbacks will be called in the context of any of the threads that called `io_service::run()`; this also means that if you call `io_service::run()` in only one thread, all callbacks are called in the context of that thread.

- **socket:** The `socket` classes are not thread-safe. Thus, you should avoid doing such as reading from a socket in one thread and write to it in a different thread (this isn't recommended in general, let alone with Boost.Asio).
- **utility:** For the `utility` classes, it usually does not make sense to be used in several threads, **nor are they thread-safe**. Most of them are meant to just be used for a short time, then go out of scope.

The Boost.Asio library itself can use several threads besides your own, but it guarantees that from those threads, it will not call any of your code. **This in turn means that callbacks are called only from the threads that have called `io_service::run()`.**

Not just networking

Boost.Asio, in addition to networking, provides other input/output facilities.

Boost.Asio allows waiting for signals, such as `SIGTERM` (software terminate), `SIGINT` (signal interrupt), `SIGSEGV` (segment violation), and so on.

You create a `signal_set` instance, and specify what signals to asynchronously wait for, and when any of them happen, your asynchronous handler is called:

```
void signal_handler(const boost::system::error_code & err, int signal)
{
    // log this, and terminate application
}

boost::asio::signal_set sig(service, SIGINT, SIGTERM);
sig.async_wait(signal_handler);
```

If `SIGINT` is generated, you'll catch it in your `signal_handler` callback.

Using Boost.Asio, you can easily connect to a serial port. The port name is `COM7` on Windows, or `/dev/ttyS0` on POSIX platforms:

```
io_service service;
serial_port sp(service, "COM7");
```

Once opened, you can set some options, such as port's baud rate, parity, stop bits, as set in the following code snippet:

```
serial_port::baud_rate rate(9600);
sp.set_option(rate);
```


Once the port is open, you can treat the serial port as a stream, and on top of that, use the free functions to read from and/or write to the serial port, such as `read()`, `async_read()`, `write`, `async_write()`, as used in the following code snippet:

```
char data[512];
read(sp, buffer(data, 512));
```

Boost.Asio also allows you to connect to Windows files, and again use the free functions, such as `read()`, `async_read()`, and so on, as used in the following code snippet:

```
HANDLE h = ::OpenFile(...);
windows::stream_handle sh(service, h);
char data[512];
read(h, buffer(data, 512));
```

You can do the same with POSIX file descriptors, such as pipes, standard I/O, various devices (but not with regular files), as done in the following code snippet:

```
posix::stream_descriptor sd_in(service, ::dup(STDIN_FILENO));
char data[512];
read(sd_in, buffer(data, 512));
```

Timers

Some I/O operations can have a deadline to complete. You can apply this only to asynchronous operations (synchronous means blocking, thus, no deadline). For instance, the next message from your partner needs to reach you in 100 milliseconds:

```
bool read = false;
void deadline_handler(const boost::system::error_code &) {
    std::cout << (read ? "read successfully" : "read failed") <<
    std::endl;
}
void read_handler(const boost::system::error_code &) {
    read = true;
}

ip::tcp::socket sock(service);
...
read = false;
char data[512];
```

```
sock.async_read_some(buffer(data, 512));
deadline_timer t(service, boost::posix_time::milliseconds(100));
t.async_wait(&deadline_handler);
service.run();
```

In the preceding code snippet, if we read our data before the deadline, `read` is set to `true`, thus our partner reached us in time. Otherwise, when `deadline_handler` is called, `read` is still set to `false`, which means we did not meet our deadline.

Boost.Asio allows for synchronous timers as well, but they are usually equivalent to a simple sleep operation. The `boost::this_thread::sleep(500)`; code and the following snippet of code accomplish the same thing:

```
deadline_timer t(service, boost::posix_time::milliseconds(500));
t.wait();
```

The `io_service` class

You've already seen that most code that uses Boost.Asio will use some instance of `io_service`. **The `io_service` is the most important class in the library**; it deals with the operating system, waiting for all asynchronous operations to end, and then calling the completion handler for each such operation.

If you choose to create your application synchronously, you won't need to worry about what I'm about to show you in this section.

You can use `io_service` instances in several ways. In the following examples, we have three asynchronous operations, two socket connections and a timer wait:

- Single-thread with one `io_service` and one handler thread:

```
io_service service_;
// all the socket operations are handled by service_
ip::tcp::socket sock1(service_);
// all the socket operations are handled by service_
ip::tcp::socket sock2(service_);
sock1.async_connect( ep, connect_handler);
sock2.async_connect( ep, connect_handler);
deadline_timer t(service_, boost::posix_time::seconds(5));
t.async_wait(timeout_handler);
service_.run();
```

- Multi-threaded with a single `io_service` instance and several handler threads:

```
io_service service_;
ip::tcp::socket sock1(service_);
ip::tcp::socket sock2(service_);
sock1.async_connect( ep, connect_handler);
sock2.async_connect( ep, connect_handler);
deadline_timer t(service_, boost::posix_time::seconds(5));
t.async_wait(timeout_handler);

for ( int i = 0; i < 5; ++i)
    boost::thread( run_service);

void run_service() {
    service_.run();
}
```

- Multi-threaded with several `io_service` instances and several threads:

```
io_service service_[2];
ip::tcp::socket sock1(service_[0]);
ip::tcp::socket sock2(service_[1]);
sock1.async_connect( ep, connect_handler);
sock2.async_connect( ep, connect_handler);
deadline_timer t(service_[0], boost::posix_time::seconds(5));
t.async_wait(timeout_handler);

for ( int i = 0; i < 2; ++i)
    boost::thread( boost::bind(run_service, i));

void run_service(int idx) {
    service_[idx].run();
}
```

First off, notice you can't have several `io_service` instances and one thread. It would make no sense to have the following code snippet:

```
for ( int i = 0; i < 2; ++i)
    service_[i].run();
```

The preceding code snippet makes no sense, because `service_[1].run()` would need `service_[0].run()` to complete first. Thus, all asynchronous operations handled by `service_[1]` would have to wait, which is not a good idea.

In all the three preceding scenarios, we're waiting for three asynchronous operations to complete. To explain the differences, we'll assume that, after a while, operation 1 completes, and just after that, operation 2 completes. We'll also assume that each completion handler takes a second to complete.

In the first case, we're waiting for all three operations to complete in one thread. Once operation 1 completes, we call its completion handler. Even though operation 2 completes just after, the completion handler for operation 2 will be called one second after the operation 1's handler completes.

In the second case, we're waiting for the three operations to complete in two threads. Once operation 1 completes, we call its completion handler in the first thread. Once operation 2 completes, just after, we'll call its completion handler instantly, in the second thread (while thread 1 is busy responding to operation 1 handler's, thread 2 is free to answer any incoming new operation).

In the third case, in case operation 1 is connect of `sock1`, and operation 2 is connect of `sock2`, the application will behave like in the second case. Thread 1 will handle connect of `sock1` completion handler, and thread 2 will handle connect of `sock2` completion handler. However, if connect of `sock1` is operation 1, and timeout of `deadline_timer t` is operation 2, thread 1 will end up handling connect of `sock1` completion handler. Therefore, timeout of `deadline_timer t` completion handler will have to wait until connect of `sock1` completion handler ends (it will wait one second), since thread 1 handles both connection handler `sock1` and timeout handler of `t`.

Here's what you should have learnt from the previous examples:

- **Situation 1 is for very basic applications.** You will always run into bottleneck problem if several handlers need to be called at the same time, as they will be called in a serial manner. If one handler takes too long to complete, all subsequent handlers will have to wait.
- **Situation 2 is for most applications.** It is very robust – if several handlers are to be called at the same time (this is possible) they will each be called in their own thread. The only bottleneck you can have is if all handler threads are busy and new handlers are to be called at that time. However, as a quick solution, just increase the number of handler threads.

- **Situation 3 is the most complex and most flexible.** You should use this only when situation 2 is not enough. That will probably be when you have thousands of concurrent (socket) connections. You can consider that each handler thread (thread running `io_service::run()`) has its own `select/epoll` loop; it waits for any socket, it monitors to have a read/write operation, and then once it finds such an operation, it executes it. In most cases, you don't need to worry about this, as you'll only need to worry if the number of sockets you're monitoring grows exponentially (greater than 1,000 sockets). In that case, having several `select/epoll` loops can increase response times.

If, in your application, you think, you'll ever need to switch to situation 3, make sure that the monitor for operations code (the code that calls `io_service::run()`) is insulated from the rest of the application, so you can easily change it.

Finally, always remember that `.run()` will always end if there are no more operations to monitor, as given in the following code snippet:

```
io_service service_;
tcp::socket sock(service_);
sock.async_connect( ep, connect_handler);
service_.run();
```

In the previous case, once `sock` has established a connection, `connect_handler` will be called, and afterwards, `service.run()` will complete.

If you want to make sure `service_.run()` continues to run, you have to assign more work to it. There are two ways of accomplishing this. One way is to assign more work inside `connect_handler` by starting another asynchronous operation.

The other way is to simulate some work for it, by using the following code snippet:

```
typedef boost::shared_ptr<io_service::work> work_ptr;
work_ptr dummy_work(new io_service::work(service_));
```

The preceding code will make sure that `service_.run()` never stops unless you either use `service_.stop()` or `dummy_work.reset(0);` // destroy `dummy_work`.

Summary

Boost.Asio is a complex library, making networking quite simple. Building it is easy. It's done quite a good job at avoiding use of macros; it's got a few macros to turn options on/off, but there's only quite a few you need to worry about.

Boost.Asio allows for both synchronous and asynchronous programming. They are very different; you should choose one way or the other as early as possible, since switching is quite complicated and prone to error.

If you go synchronous, you can choose between exceptions and error codes, going from exceptions to error codes is simple; just add one more argument to the function call (the error code).

Boost.Asio is not just for networking. It's got a few more features, making it even more valuable, such as signals, timers, and so on.

In the next chapter, we'll delve into the multitude of functions and classes Boost.Asio provides for networking. Also, we'll learn a few tricks about asynchronous programming.

2

Boost.Asio Fundamentals

In this chapter, we'll cover what you definitely need to know when using Boost.Asio. We'll delve deeper into asynchronous programming, which is trickier than synchronous and is much more fun.

The Network API

This section shows what you definitely need to know in order to write a networking application using Boost.Asio.

Boost.Asio namespaces

Everything in Boost.Asio resides in the `boost::asio` namespace, or a sub-namespace of that:

- `boost::asio`: This is where core classes and functions reside. The important classes are `io_service` and `streambuf`. Here, we also have the free functions, such as `read`, `read_at`, `read_until`, their asynchronous counterparts, and their write and asynchronous write counterparts.
- `boost::asio::ip`: This is where the networking part resides. The important classes are `address`, `endpoint`, `tcp`, `udp`, `icmp`, and the important free functions are `connect` and `async_connect`. Note that in the `boost::asio::ip::tcp::socket` name, `socket` is just a typedef keyword inside the `boost::asio::ip::tcp` class.
- `boost::asio::error`: This namespace contains the error codes you can get while calling I/O routines.
- `boost::asio::ssl`: This namespace contains classes dealing with SSL.
- `boost::asio::local`: This namespace contains POSIX-specific classes.
- `boost::asio::windows`: This namespace contains Windows-specific classes.

IP addresses

To deal with IP addresses, Boost.Asio provides the `ip::address`, `ip::address_v4` and `ip::address_v6` classes.

They offer quite a few functions. Here are the most important ones:

- `ip::address(v4_or_v6_address)`: This function converts a v4 or v6 address to `ip::address`
- `ip::address::from_string(str)`: This function creates an address from an IPv4 address (separated by dots) or from an IPv6 (hexadecimal notation)
- `ip::address::to_string()`: This function returns the friendly representation of the address
- `ip::address_v4::broadcast([addr, mask])`: This function creates a broadcast address
- `ip::address_v4::any()`: This function returns an address that represents any address
- `ip::address_v4::loopback()`, `ip::address_v6::loopback()`: This function returns the loopback address (for v4/v6 protocol)
- `ip::host_name()`: This function returns the name of the current host as `string` datatype

You'll likely use `ip::address::from_string` most of the time:

```
ip::address addr = ip::address::from_string("127.0.0.1");
```

If you need to connect to a host name, read on. This code snippet won't work:

```
// throws an exception
ip::address addr = ip::address::from_string("www.yahoo.com");
```

Endpoints

Endpoint is an address you connect to, together with a port. Each different type of socket has its own endpoint class, such as `ip::tcp::endpoint`, `ip::udp::endpoint`, and `ip::icmp::endpoint`.

If you want to connect to localhost, port 80, here you go:

```
ip::tcp::endpoint ep( ip::address::from_string("127.0.0.1"), 80);
```

You can construct an endpoint in three ways:

- `endpoint()`: This is the default constructor and can be used sometimes for UDP/ICMP sockets
- `endpoint(protocol, port)`: This is usually used on server sockets for accepting new connections
- `endpoint(addr, port)`: This creates an endpoint to an address and a port

Its examples are:

```
ip::tcp::endpoint ep1;
ip::tcp::endpoint ep2(ip::tcp::v4(), 80);
ip::tcp::endpoint ep3(ip::address::from_string("127.0.0.1"), 80);
```

If you want to connect to a hostname (not an IP address), here's what you do:

```
// outputs "87.248.122.122"
io_service service;
ip::tcp::resolver resolver(service);
ip::tcp::resolver::query query("www.yahoo.com", "80");
ip::tcp::resolver::iterator iter = resolver.resolve(query);
ip::tcp::endpoint ep = *iter;
std::cout << ep.address().to_string() << std::endl;
```

You'll replace `tcp` with the socket type you need. First, create a query for the name you want, then resolve it using the `resolve()` function. If successful, it will return at least one entry. On the given returned iterator, either always use only the first entry, or iterate through the list.

Given an endpoint, you can obtain its address, port, and IP protocol (v4 or v6):

```
std::cout << ep.address().to_string() << ":" << ep.port()
          << "/" << ep.protocol() << std::endl;
```

Sockets

Boost.Asio comes with three types of socket classes: `ip::tcp`, `ip::udp`, and `ip::icmp`, and is of course extensible. You can create your own socket class, even though that is pretty complicated. In case you choose to do so, take a look at `boost/asio/ip/tcp.hpp`, `boost/asio/ip/udp.hpp`, and `boost/asio/ip/icmp.hpp`. They are all pretty small classes with internal typedef keywords.

You can think of the `ip::tcp`, `ip::udp`, `ip::icmp` classes as placeholders; they give you easy access to other classes/functions, which are given as follows:

- `ip::tcp::socket`, `ip::tcp::acceptor`, `ip::tcp::endpoint`,
`ip::tcp::resolver`, `ip::tcp::iostream`
- `ip::udp::socket`, `ip::udp::endpoint`, `ip::udp::resolver`
- `ip::icmp::socket`, `ip::icmp::endpoint`, `ip::icmp::resolver`

The `socket` classes create a corresponding socket. You always pass the `io_service` instance at construction:

```
io_service service;  
ip::udp::socket sock(service)  
sock.set_option(ip::udp::socket::reuse_address(true));
```

Each of the socket names is a `typedef` keyword:

- `ip::tcp::socket = basic_stream_socket<tcp>`
- `ip::udp::socket = basic_datagram_socket<udp>`
- `ip::icmp::socket = basic_raw_socket<icmp>`

Synchronous error codes

All synchronous functions have overloads that either throw an exception or return an error code, as given in the following code snippet:

```
sync_func( arg1, arg2 ... argN); // throws  
boost::system::error_code ec;  
sync_func( arg1 arg2, ..., argN, ec); // returns error code
```

In the remainder of this chapter, you'll see a lot of synchronous functions. To keep things simple, I omitted showing the overloads that return an error code, but they exist.

Socket member functions

The functions are split into a few groups. Not all functions are available for each type of socket. A list at the end of this section will show you which function belongs to which socket classes.

Note that all asynchronous functions return immediately, while their synchronous counterparts will return only after the operation has been completed.

Connecting-related functions

These are the functions that connect or bind the socket, disconnect it, and query whether the connection is active or not:

- `assign(protocol, socket)`: This function assigns a raw (native) socket to this socket instance. Use it when dealing with legacy code (that is, the native sockets are already created).
- `open(protocol)`: This function opens a socket with the given IP protocol (v4 or v6). You'll use this mainly for UDP/ICMP sockets, or for server sockets.
- `bind(endpoint)`: This function binds to this address.
- `connect(endpoint)`: This function synchronously connects to the address.
- `async_connect(endpoint)`: This function asynchronously connects to the address.
- `is_open()`: This function returns `true` if the socket is open.
- `close()`: This function closes the socket. Any asynchronous operations on this socket are canceled immediately and will complete with `error::operation_aborted` error code.
- `shutdown(type_of_shutdown)`: This function disables send operations, receive operations, or both, starting now.
- `cancel()`: This function cancels all asynchronous operations on this socket. The asynchronous operations on this socket will all finish immediately with the `error::operation_aborted` error code.

Its example is given as follows:

```
ip::tcp::endpoint ep( ip::address::from_string("127.0.0.1"), 80);
ip::tcp::socket sock(service);
sock.open(ip::tcp::v4());      n
sock.connect(ep);
sock.write_some(buffer("GET /index.html\r\n"));
char buff[1024]; sock.read_some(buffer(buff,1024));
sock.shutdown(ip::tcp::socket::shutdown_receive);
sock.close();
```

Read/write functions

These are the functions that perform the I/O on the socket.

For asynchronous functions, the signature of the handler, `void handler (const boost::system::error_code& e, size_t bytes)`, is the same:

- `async_receive(buffer, [flags,] handler)`: This function starts the asynchronous receive operation of data from the socket.
- `async_read_some(buffer, handler)`: This function is equivalent to `async_receive(buffer, handler)`.
- `async_receive_from(buffer, endpoint[, flags], handler)`: This function starts the asynchronous receive of data from a specific endpoint.
- `async_send(buffer[, flags], handler)`: This function starts an asynchronous send function of the buffer's data.
- `async_write_some(buffer, handler)`: This function is equivalent to `async_send(buffer, handler)`.
- `async_send_to(buffer, endpoint, handler)`: This function starts an asynchronous send function of the buffer's data to the specific endpoint.
- `receive(buffer[, flags])`: This function synchronously receives data in the given buffer. The function blocks until data is received, or an error occurs.
- `read_some(buffer)`: This function is equivalent to `receive(buffer)`.
- `receive_from(buffer, endpoint[, flags])`: This function synchronously receives data from a given endpoint into the given buffer. The function blocks until data is received, or an error occurs.
- `send(buffer[, flags])`: This function synchronously sends the buffer's data. The function blocks until data is successfully sent, or an error occurs.
- `write_some(buffer)`: This function is equivalent to `send(buffer)`.
- `send_to(buffer, endpoint[, flags])`: This function synchronously sends the buffer's data to a given endpoint. The function blocks until data is successfully sent or an error occurs.
- `available()`: This function returns how many bytes can be read synchronously without blocking.

We'll talk about buffers shortly. Let's examine the flags. The default value for flags is 0 but can be a combination of:

- `ip::socket_type::socket::message_peek`: This flag only peeks at the message. It will return the message, but the next call to read the message will re-read this message.
- `ip::socket_type::socket::message_out_of_band`: This flag processes **out-of-band (OOB)** data. OOB data is data that is flagged as more important than normal data. A discussion about OOB data is out of the scope of this book.
- `ip::socket_type::socket::message_do_not_route`: This flag specifies that the message should be sent without using routing tables.
- `ip::socket_type::socket::message_end_of_record`: This flag specifies that the data marks the end of a record. This is not supported on Windows.

You will most likely use `message_peek`, if ever you use the following code snippet:

```
char buff[1024];
sock.receive(buffer(buff), ip::tcp::socket::message_peek );
memset(buff, 0, 1024);
// re-reads what was previously read
sock.receive(buffer(buff) );
```

Following are examples that give guidance to read synchronously and asynchronously to different types of sockets:

- Example 1 is to write and read synchronously to a TCP socket:


```
ip::tcp::endpoint ep( ip::address::from_string("127.0.0.1"), 80);
ip::tcp::socket sock(service);
sock.connect(ep);
sock.write_some(buffer("GET /index.html\r\n"));
std::cout << "bytes available " << sock.available() << std::endl;
char buff[512];
size_t read = sock.read_some(buffer(buff));
```
- Example 2 is to read and write synchronously to a UDP socket:


```
ip::udp::socket sock(service);
sock.open(ip::udp::v4());
ip::udp::endpoint receiver_ep("87.248.112.181", 80);
sock.send_to(buffer("testing\n"), receiver_ep);
char buff[512];
ip::udp::endpoint sender_ep;
sock.receive_from(buffer(buff), sender_ep);
```



Note that to read from a UDP socket using `receive_from`, you need a default-constructed endpoint, as shown in the previous code snippet.

- Example 3 is to read asynchronously from a UDP server socket:

```
using namespace boost::asio;
io_service service;
ip::udp::socket sock(service);
boost::asio::ip::udp::endpoint sender_ep;
char buff[512];

void on_read(const boost::system::error_code & err, std::size_t
read_bytes) {
    std::cout << "read " << read_bytes << std::endl;
    sock.async_receive_from(buffer(buff), sender_ep, on_read);
}

int main(int argc, char* argv[]) {
    ip::udp::endpoint ep( ip::address::from_string("127.0.0.1"),
8001);
    sock.open(ep.protocol());
    sock.set_option(boost::asio::ip::udp::socket::reuse_
address(true));
    sock.bind(ep);
    sock.async_receive_from(buffer(buff,512), sender_ep, on_read);
    service.run();
}
```

Socket control

These functions deal with the advanced socket options:

- `get_io_service()`: This function returns the `io_service` instance that was passed at construction
- `get_option(option)`: This function returns a socket option
- `set_option(option)`: This function sets a socket option
- `io_control(cmd)`: This function executes an I/O command on the socket

Here are the options you can get/set for a socket:

Name	Description	Type
<code>broadcast</code>	If <code>true</code> , it allows broadcasting messages	<code>bool</code>
<code>debug</code>	If <code>true</code> , it enables socket-level debugging	<code>bool</code>
<code>do_not_route</code>	If <code>true</code> , it prevents routing and use local interfaces only	<code>bool</code>
<code>enable_connection_aborted</code>	If <code>true</code> , it reports connections that were aborted on <code>accept()</code>	<code>bool</code>
<code>keep_alive</code>	If <code>true</code> , it sends keep-alives	<code>bool</code>
<code>linger</code>	If <code>true</code> , socket lingers on <code>close()</code> if there's unsent data	<code>bool</code>
<code>receive_buffer_size</code>	This is a received buffer size for a socket	<code>int</code>
<code>receive_low_watermark</code>	This provides a minimum number of bytes to process for socket input	<code>int</code>
<code>reuse_address</code>	If <code>true</code> , socket can be bound to an address already in use	<code>bool</code>
<code>send_buffer_size</code>	This sends buffer size for a socket	<code>int</code>
<code>send_low_watermark</code>	This provides a minimum number of bytes to send for socket output	<code>int</code>
<code>ip::v6_only</code>	If <code>true</code> , it allows only IPv6 communication	<code>bool</code>

Each name represents an inner socket typedef or a class. Here is how to use them:

```
ip::tcp::endpoint ep( ip::address::from_string("127.0.0.1"), 80);
ip::tcp::socket sock(service);
sock.connect(ep);
// TCP socket can reuse address
ip::tcp::socket::reuse_address ra(true);
sock.set_option(ra);
// get sock receive buffer size
ip::tcp::socket::receive_buffer_size rbs;
sock.get_option(rbs);
std::cout << rbs.value() << std::endl;
// set sock's buffer size to 8192
ip::tcp::socket::send_buffer_size sbs(8192);
sock.set_option(sbs);
```




The socket needs to be open for the previous features to work, otherwise, an exception is thrown.

TCP versus UDP versus ICMP

As I previously said, not all member functions are available to all `socket` classes. I've made a list where member functions differ. If a member function is not here, it means that it's present in all socket classes:

Name	TCP	UDP	ICMP
<code>async_read_some</code>	Yes	-	-
<code>async_receive_from</code>	-	Yes	Yes
<code>async_write_some</code>	Yes	-	-
<code>async_send_to</code>	-	Yes	Yes
<code>read_some</code>	Yes	-	-
<code>receive_from</code>	-	Yes	Yes
<code>write_some</code>	Yes	-	-
<code>send_to</code>	-	Yes	Yes

Miscellaneous functions

Other functions unrelated to connection or input/output are as follows:

- `local_endpoint()`: This function returns the address, where the socket is connected locally.
- `remote_endpoint()`: This function returns the remote address, where the socket is connected to.
- `native_handle()`: This function returns the handle of the raw socket. You only need this if you want to call a raw function not supplied by Boost.Asio.
- `non_blocking()`: This function returns `true` if the socket is non-blocking, `false` otherwise.
- `native_non_blocking()`: This function returns `true` if the socket is non-blocking, `false` otherwise. However, it will call the native API on the raw socket. Usually, you don't need this (`non_blocking()` already caches this result); you should only use it if you deal with the `native_handle()` directly yourself.
- `at_mark()`: This function returns `true` if the socket is about to read OOB data. You will very seldom need this.

Other considerations

As a final note, a socket instance cannot be copied, as the Copy constructor and operator = are inaccessible:

```
ip::tcp::socket s1(service), s2(service);
s1 = s2; // compile time error
ip::tcp::socket s3(s1); // compile time error
```

This makes a lot of sense, since each instance holds and manages a resource (the raw socket itself). If we were to allow copy-construction, we could end up with two instances having the same raw socket; they would need to somehow manage ownership (either one instance has ownership, or use reference counting, or any method). Boost.Asio chose to disallow copying (if you want to create copies, just use a shared pointer):

```
typedef boost::shared_ptr<ip::tcp::socket> socket_ptr;
socket_ptr sock1(new ip::tcp::socket(service));
socket_ptr sock2(sock1); // ok
socket_ptr sock3;
sock3 = sock1; // ok
```

Socket buffers

When reading from or writing to a socket, you'll need a buffer, one that will hold the incoming data or the outgoing data. The memory in the buffer must outlive the I/O operation; you have to make sure it is not deallocated or goes out of scope as long as the I/O operation lasts.

This is extremely easy for synchronous operations; of course, the `buff` will outlive both receive and send:

```
char buff[512];
...
sock.receive(buffer(buff));
strcpy(buff, "ok\n");
sock.send(buffer(buff));
```

This is not so straightforward for asynchronous operations, as given in the following code snippet:

```
// very bad code ...
void on_read(const boost::system::error_code & err, std::size_t read_
bytes)
{ ... }
void func() {
    char buff[512];
    sock.async_receive(buffer(buff), on_read);
}
```

After the call to `async_receive()`, `buff` will go out of scope, thus its memory will be deallocated. When we're about to actually receive some data on the socket, we'll copy them into memory we don't own anymore; it could either be deallocated, or reallocated by some other code for some other data, thus, corrupting memory.

There are several solutions to the above problem:

- Use global buffers
- Create a buffer, and destroy it when the operation completes
- Have a connection object that maintains the socket, and additional data, such as, buffer(s)

The first solution is not that good, since we all know global variables are quite bad. Besides, what happens if two handlers use the same buffer?

Here's how you can implement the second solution:

```
void on_read(char * ptr, const boost::system::error_code & err,
std::size_t read_bytes) {
    delete[] ptr;
}
....
char * buff = new char[512];
sock.async_receive(buffer(buff, 512), boost::bind(on_
read,buff,_1,_2));
```

Or, if you want the buffer to go automatically out of scope as the operation completes, use a shared pointer:

```

struct shared_buffer {
    boost::shared_array<char> buff;
    int size;
    shared_buffer(size_t size) : buff(new char[size]), size(size) {
    }
    mutable_buffers_1 asio_buff() const {
        return buffer(buff.get(), size);
    }
};

// when on_read goes out of scope, the boost::bind object is released,
// and that will release the shared_buffer as well
void on_read(shared_buffer, const boost::system::error_code & err,
             std::size_t read_bytes) {}

...
shared_buffer buff(512);
sock.async_receive(buff.asio_buff(), boost::bind(on_read,buff,_1,_2));

```

The `shared_buffer` class holds internally `shared_array<>`, which are the copies of the `shared_buffer` instance that `shared_array<>` will stay alive for – when the last one goes out of scope, `shared_array<>` is automatically destroyed, which is just what we want.

This works as you'd expect, because Boost.Asio will keep a copy to the completion handler, to call when the operation completes. That copy is a `boost::bind` functor, which internally holds a copy to our `shared_buffer` instance. It is pretty neat!

The third option is to use a connection object that maintains the socket and additional data, such as the buffers, is usually the right solution but is pretty complex. It will be discussed at the end of this chapter.

The buffer function wrapper

Throughout the code you've seen that whenever we need a buffer for a read/write operation, the code wraps the real buffer object into a `buffer()` call, and passes it to the function:

```

char buff[512];
sock.async_receive(buffer(buff), on_read);

```

This basically wraps any buffer we have into a class that allows the Boost.Asio functions to iterate through the buffer. Say, you use the following code:

```

sock.async_receive(some_buffer, on_read);

```

The `some_buffer` instance needs to meet some requirements, namely `ConstBufferSequence` or `MutableBufferSequence` (you can look them up in Boost.Asio's documentation). The details of creating your own class to meet these requirements are pretty complex, but Boost.Asio already provides some classes, modeling those requirements. You don't access them directly, you use the `buffer()` function.

Suffice to say, you can wrap any of the following into a `buffer()` function:

- A `char[]` const array
- A `void*` pointer and size in characters
- An `std::string` string
- An `POD[]` const array (POD stands for plain old data, meaning, constructor and destructor do nothing)
- An `std::vector` array of any POD
- A `boost::array` array of any POD
- An `std::array` array of any POD

The following code works:

```
struct pod_sample { int i; long l; char c; };
...
char b1[512];
void * b2 = new char[512];
std::string b3; b3.resize(128);
pod_sample b4[16];
std::vector<pod_sample> b5; b5.resize(16);
boost::array<pod_sample,16> b6;
std::array<pod_sample,16> b7;
sock.async_send(buffer(b1), on_read);
sock.async_send(buffer(b2,512), on_read);
sock.async_send(buffer(b3), on_read);
sock.async_send(buffer(b4), on_read);
sock.async_send(buffer(b5), on_read);
sock.async_send(buffer(b6), on_read);
sock.async_send(buffer(b7), on_read);
```

All in all, rather than creating your own class to meet the requirements of `ConstBufferSequence` or `MutableBufferSequence`, you can probably create a class that will hold the buffer as long as it's needed, and return an instance of `mutable_buffers_1`, which is what we did in `shared_buffer` class earlier.

The read/write/connect free functions

Boost.Asio gives you free functions to deal with I/O. I've split them into four groups.

The connect functions

These functions connect the socket to an endpoint:

- `connect(socket, begin [, end] [, condition])`: This function synchronously connects by trying each endpoint in the sequence `begin` and `end`. The `begin` iterator is the result of a `socket_type::resolver::query` call (you might want to check out the *Endpoints* section again). Specifying the `end` iterator is optional; you can forget about it. You can supply a condition function that is called before each connection attempt. Its signature is `Iterator connect_condition(const boost::system::error_code &err, Iterator next) ;`. You can choose to return a different iterator than `next`, allowing you to skip over some endpoints.
- `async_connect(socket, begin [, end] [, condition], handler)`: This function executes the connection asynchronously, and at the end, it calls the completion handler. The handler's signature is `void handler(const boost::system::error_code &err, Iterator iterator) ;`. The second parameter passed to the handler is the successfully connected endpoint (or the `end` iterator otherwise).

Its example is given as follows:

```
using namespace boost::asio::ip;
tcp::resolver resolver(service);
tcp::resolver::iterator iter = resolver.resolve(
    tcp::resolver::query("www.yahoo.com",
    "80"));
tcp::socket sock(service);
connect(sock, iter);
```

A hostname can resolve into more than one address, thus `connect` and `async_connect` release you of the burden of trying each address until one works; they do that for you.

The read/write functions

These functions read from or write to a stream (which can be a socket, or any other class that behaves like a stream):

- `async_read(stream, buffer [, completion] , handler)`: This function asynchronously reads from a stream. On completion, the handler is called. The handler's signature is `void handler(const boost::system::error_code &err, size_t bytes)`; . You can optionally specify a completion function. The completion function is called after each successful read, and tells Boost.Asio if the `async_read` operation is complete (if not, it will continue to read). Its signature is `size_t completion(const boost::system::error_code&err, size_t bytes_transferred)`. When this completion function returns 0, we consider the read operation complete; if it returns a non-zero value, it indicates the maximum number of bytes to be read on the next call to the stream's `async_read_some` operation. An example will follow to clear up this.
- `async_write(stream, buffer [, completion] , handler)`: This function asynchronously writes to a stream. The meaning of the arguments is similar to `async_read`.
- `read(stream, buffer [, completion])`: This function synchronously reads from a stream. The meaning of the arguments is similar to `async_read`.
- `write(stream, buffer [, completion])`: This function synchronously writes to a stream. The meaning of the arguments is similar to `async_read`:
 - `async_read(stream, stream_buffer [, completion] , handler)`
 - `async_write(stream, stream_buffer [, completion] , handler)`
 - `write(stream, stream_buffer [, completion])`
 - `read(stream, stream_buffer [, completion])`

First, note that instead of socket, the first argument is a stream. This includes sockets but is not limited. For instance, instead of a socket, you can use a Windows file handle.

Each read or write operation will end when one of these conditions occur:

- The supplied buffer is full (for read) or all the data in the buffer has been written (for write)
- The completion function returns 0 (if you supplied one such function)
- An error occurs

The following code will asynchronously read from the socket until it meets '\n':

```

io_service service;
ip::tcp::socket sock(service);
char buff[512];
int offset = 0;
size_t up_to_enter(const boost::system::error_code &, size_t bytes) {
    for ( size_t i = 0; i < bytes; ++i)
        if ( buff[i + offset] == '\n')
            return 0;
    return 1;
}
void on_read(const boost::system::error_code &, size_t) {}
...
async_read(sock, buffer(buff), up_to_enter, on_read);

```

Boost.Asio comes with a few helper completion functors as well:

- `transfer_at_least(n)`
- `transfer_exactly(n)`
- `transfer_all()`

Its example is given as follows:

```

char buff[512];
void on_read(const boost::system::error_code &, size_t) {}
// read exactly 32 bytes
async_read(sock, buffer(buff), transfer_exactly(32), on_read);

```

The last four functions, instead of the usual `buffer`, use a `stream_buffer` function, which is the Boost.Asio's `std::streambuf` derived class. STL streams and stream buffers are very flexible; here's an example:

```

io_service service;
void on_read(streambuf& buf, const boost::system::error_code &,
size_t) {
    std::istream in(&buf);
    std::string line;
    std::getline(in, line);
    std::cout << "first line: " << line << std::endl;
}
int main(int argc, char* argv[]) {
    HANDLE file = ::CreateFile("readme.txt", GENERIC_READ, 0, 0,
        OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL | FILE_FLAG_OVERLAPPED,
0);
    windows::stream_handle h(service, file);
    streambuf buf;
    async_read(h, buf, transfer_exactly(256),
        boost::bind(on_read, boost::ref(buf), _1, _2));
    service.run();
}

```


Here, I've shown you that you can also call `async_read` (and the like) on a Windows file handle. We read the first 256 characters, and store them into the buffer. When the read operation is complete, `on_read` is called, I create `std::istream` passing the buffer, read the first line (`std::getline`), and dump it to the console.

The `read_until/async_read_until` functions

These functions read until a condition is met:

- `async_read_until(stream, stream_buffer, delim, handler)`: This function starts an asynchronous read operation. The read operation will stop when a delimiter is met. The delimiter can be any of a character, `std::string` or `boost::regex`. The handler's signature is `void handler(const boost::system::error_code &err, size_t bytes);`.
- `async_read_until(stream, stream_buffer, completion, handler)`: This function is the same as the previous one, but instead of a delimiter, we have a completion function. The completion's signature is `pair<iterator, bool> completion(iterator begin, iterator end);`, where `iterator = is_buffers_iterator<streambuf::const_buffers_type>`. What you need to remember is that the iterator is of type `random-access-iterator`. You scan the range `(begin, end)`, and decide if the read operation should stop or not. You will return a pair; the first member will be an iterator passed at the end of the last character consumed by the function; the second member is `true` if the read operation should stop, or `false` otherwise.
- `read_until(stream, stream_buffer, delim)`: This function performs a synchronous read operation. The parameters' meaning is same as in `async_read_until`.
- `read_until(stream, stream_buffer, completion)`: This function performs a synchronous read operation. The parameters' meaning is same as in `async_read_until`.

The following example will read up to a punctuation sign:

```
typedef buffers_iterator<streambuf::const_buffers_type> iterator;
std::pair<iterator, bool> match_punct(iterator begin, iterator end) {
    while ( begin != end)
        if ( std::ispunct(*begin) )
            return std::make_pair(begin, true);
    return std::make_pair(end, false);
}
```

```
void on_read(const boost::system::error_code &, size_t) {}
...
streambuf buf;
async_read_until(sock, buf, match_punct, on_read);
```

If we wanted to read up to space, we'd modify the last line to:

```
async_read_until(sock, buff, ' ', on_read);
```

The *_at functions

These functions do random read/write operations on a stream. You specify where the read or write operation is to start from (the offset):

- `async_read_at(stream, offset, buffer [, completion], handler):` This function starts an asynchronous read operation starting at offset, on the given stream. When the operation completes, it will call the handler. The handler's signature is `void handler(const boost::system::error_code& err, size_t bytes);`. The buffer can be the usual `buffer()` wrapper or a `streambuf` function. If you specify a completion function, it is called after each successful read, and tells Boost.Asio if the `async_read_at` operation is complete (if not, it will continue to read). Its signature is `size_t completion(const boost::system::error_code& err, size_t bytes);`. When this completion function returns 0, we consider the read operation complete; if it returns a non-zero value, it indicates the maximum number of bytes to be read on the next call to the stream's `async_read_some_at`.
- `async_write_at(stream, offset, buffer [, completion], handler):` This function starts an asynchronous write operation. The parameters' meaning is the same as `async_read_at`.
- `read_at(stream, offset, buffer [, completion]):` This function reads at offset, on the given stream. The parameters' meaning is the same as `async_read_at`.
- `write_at(stream, offset, buffer [, completion]):` This function writes at offset, on the given stream. The parameters' meaning is the same as `async_read_at`.

These functions do not deal with sockets. They deal with random access streams; in other words, streams that can be accessed randomly. Sockets are clearly not the case (sockets are forward-only).

Here's how you can read 128 bytes from a file, starting at offset 256:

```
io_service service;
int main(int argc, char* argv[]) {
    HANDLE file = ::CreateFile("readme.txt", GENERIC_READ, 0, 0,
        OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL | FILE_FLAG_OVERLAPPED,
    0);
    windows::random_access_handle h(service, file);
    streambuf buf;
    read_at(h, 256, buf, transfer_exactly(128));
    std::istream in(&buf);
    std::string line;
    std::getline(in, line);
    std::cout << "first line: " << line << std::endl;
}
```

Asynchronous programming

This section delves very deep into some of the issues you'll run into when doing asynchronous programming. After reading it once, I suggest you come back to it, as you progress through the book, to solidify your understanding of these concepts.

The need for going asynchronous

As I previously said, usually, synchronous work is quite easier than asynchronous programming. This is because, it's much easier to think linearly (call to function A, call to function A ends, call to function B, call to function B ends, and so on, so that to think in event-handling manner). In the latter case, you can have, lets say, five events, and you can never know the order in which they execute, and you can't even know if they will all execute!

Even though asynchronous programming is harder, you'll very likely prefer it, in say, writing servers that need to deal with lots of concurrent clients. The more concurrent clients you have, the easier asynchronous programming is compared to synchronous programming.

Say, you have an application that deals with 1,000 concurrent clients, each message client to server and server to client, ending in '`\n`'.

Synchronous code, 1 thread:

```
using namespace boost::asio;
struct client {
    ip::tcp::socket sock;
    char buff[1024]; // each msg is at maximum this size
    int already_read; // how much have we already read?
};
std::vector<client> clients;
void handle_clients() {
    while ( true)
        for ( int i = 0; i < clients.size(); ++i)
            if ( clients[i].sock.available() ) on_read(clients[i]);
}
void on_read(client & c) {
    int to_read = std::min( 1024 - c.already_read, c.sock.
available());
    c.sock.read_some( buffer(c.buff + c.already_read, to_read));
    c.already_read += to_read;
    if ( std::find(c.buff, c.buff + c.already_read, '\n') < c.buff +
c.already_read) {
        int pos = std::find(c.buff, c.buff + c.already_read, '\n') -
c.buff;
        std::string msg(c.buff, c.buff + pos);
        std::copy(c.buff + pos, c.buff + 1024, c.buff);
        c.already_read -= pos;
        on_read_msg(c, msg);
    }
}
void on_read_msg(client & c, const std::string & msg) {
    // analyze message, and write back
    if ( msg == "request_login")
        c.sock.write( "request_ok\n");
    else if ...
}
```

The one thing you want to avoid in any server (and in any networking application basically) is for code to become unresponsive. In our case, we want the `handle_clients()` function to block as little as possible. If the function blocks at any point, any incoming messages from clients will have to wait until the function unblocks and starts processing them.

In order to stay responsive, we only read from a socket when there's data on it, that is, `if (clients[i].sock.available()) on_read(clients[i])`. In `on_read`, we will only read as much as is available; calling `read_until(c.sock, buffer(...), '\n')` would be a very bad idea, since that will block until we've read the full message from this particular client (we never know when that will happen).

The bottleneck here is the `on_read_msg()` function; as long as that executes, any incoming messages are put on hold. A well-written `on_read_msg()` function will make sure that hardly happens, but it can still happen (sometimes writing to a socket can be blocked when its buffers fill up, for instance).

Synchronous code, 10 threads:

```
using namespace boost::asio;
struct client {
    // ... same as before
    bool set_reading() {
        boost::mutex::scoped_lock lk(cs_);
        if ( is_reading_ ) return false; // already reading
        else { is_reading_ = true; return true; }
    }
    void unset_reading() {
        boost::mutex::scoped_lock lk(cs_);
        is_reading_ = false;
    }
private:
    boost::mutex cs_;
    bool is_reading_;
};
std::vector<client> clients;
void handle_clients() {
    for ( int i = 0; i < 10; ++i )
        boost::thread( handle_clients_thread );
}
void handle_clients_thread() {
    while ( true )
        for ( int i = 0; i < clients.size(); ++i )
            if ( clients[i].sock.available() )
                if ( clients[i].set_reading() ) {
                    on_read( clients[i] );
                    clients[i].unset_reading();
                }
}
void on_read(client & c) {
    // same as before
}
void on_read_msg(client & c, const std::string & msg) {
    // same as before
}
```

In order to use more threads, we'll need to synchronize them, and that's what the `set_reading()` and `set_unreading()` functions do. The `set_reading()` function is very important. You want to "test for reading and mark as reading" in one step. If you had two steps ("test for reading" and "mark as reading"), you can have two threads that test for reading successfully for the same client, and then you'd end up with two threads calling `on_read` for the same client, ending up in data corruption and possibly an application crash.

You'll notice that the code becomes increasingly complex.

There's a third option for synchronized code, that is, to have one thread per client. But as the number of concurrent clients grows, this becomes pretty much a no-no situation.

And now, let's go asynchronous. We're constantly asynchronously reading. When a client asks us something, `on_read` gets called, we answer back, and then wait for the next request to come (do another asynchronous read operation).

Asynchronous code, 10 threads:

```
using namespace boost::asio;
io_service service;
struct client {
    ip::tcp::socket sock;
    streambuf buff; // reads the answer from the client
}
std::vector<client> clients;
void handle_clients() {
    for ( int i = 0; i < clients.size(); ++i)
        async_read_until(clients[i].sock, clients[i].buff, '\n',
                        boost::bind(on_read, clients[i], _1, _2));
    for ( int i = 0; i < 10; ++i)
        boost::thread(handle_clients_thread);
}

void handle_clients_thread() {
    service.run();
}

void on_read(client & c, const error_code & err, size_t read_bytes) {
    std::istream in(&c.buff);
    std::string msg;
    std::getline(in, msg);
    if ( msg == "request_login")
```

```
        c.sock.async_write( "request_ok\n", on_write);
    else if ...
    ...
    // now, wait for the next read from the same client
    async_read_until(c.sock, c.buff, '\n',
                    boost::bind(on_read, c, _1, _2));
}
```

Notice how simple the code became. The `client` structure has only two members, `handle_clients()` just calls `async_read_until`, and then it creates ten threads, each calling `service.run()`. These threads will process and dispatch any asynchronous read operations from or write operations to clients. One more thing to note is that `on_read()` will constantly prepare for the next asynchronous read operation (see the last line of code).

Asynchronous `run()`, `run_one()`, `poll()`, `poll_one()`

To implement the listening loop, `io_service` class provides four functions, such as `run()`, `run_one()`, `poll()`, and `poll_one()`. While most of the time you'll be happy with `service.run()`. You'll learn here what the other functions accomplish.

Running forever

Once again, `run()` will run as long as there are pending operations to be executed or you manually call `io_service::stop()`. To keep the `io_service` instance running, usually you add one or more asynchronous operations, and when they are executed, you keep adding asynchronous operations on and on, as in the following code:

```
using namespace boost::asio;
io_service service;
ip::tcp::socket sock(service);
char buff_read[1024], buff_write[1024] = "ok";
void on_read(const boost::system::error_code &err, std::size_t bytes)
{
    ;
}
void on_write(const boost::system::error_code &err, std::size_t bytes)
{
    sock.async_read_some(buffer(buff_read), on_read);
}
void on_read(const boost::system::error_code &err, std::size_t bytes)
{
    // ... process the read ...
}
```

```

        sock.async_write_some(buffer(buff_write,3), on_write);
    }
    void on_connect(const boost::system::error_code &err) {
        sock.async_read_some(buffer(buff_read), on_read);
    }
    int main(int argc, char* argv[]) {
        ip::tcp::endpoint ep( ip::address::from_string("127.0.0.1"),
        2001);
        sock.async_connect(ep, on_connect);
        service.run();
    }

```

When `service.run()` is called, there's one asynchronous operation pending. When the socket gets connected to a server, `on_connect` is called, which will add one more asynchronous operation. After `on_connect` finishes, we're left with one pending operation (read). When `on_read` is called, we write an answer, which adds another pending operation. When `on_read` finishes, we're left with one pending operation (write). When `on_write` is called, we read the next message from the server, which will add another pending operation. When `on_write` finishes, we have one pending operation (read). And so, the cycle continues, until we decide to close the application.

The `run_one()`, `poll()`, `poll_one()` functions

You've seen me write that asynchronous function handlers are called in threads that have previously called `io_service::run`. I've said this for simplicity, and because at least 90 to 95 percent of the time, this is the only function you will use. The same holds true for threads calling `run_one()`, `poll()`, or `poll_one()`.

The `run_one()` function will execute and dispatch at most one asynchronous operation:

- If there are no pending operations, function returns immediately, and returns 0
- If there are pending operations, function blocks until first operation is executed, and then returns 1

You can consider the following code equivalent:

```

io_service service;
service.run(); // OR
while ( !service.stopped()) service.run_once();

```


You can use `run_once()` to start an asynchronous operation, and then wait for it to complete:

```
io_service service;
bool write_complete = false;
void on_write(const boost::system::error_code & err, size_t bytes)
{ write_complete = true; }
...
std::string data = "login ok";
write_complete = false;
async_write(sock, buffer(data), on_write);
do service.run_once() while (!write_complete);
```

There are also some examples that make use of `run_one()`, bundled with Boost.Asio like `blocking_tcp_client.cpp` and `blocking_udp_client.cpp`.

The `poll_one` function runs at most one pending operation that is ready to run, without blocking:

- If there is at least one operation pending, and that is ready to be run without blocking, the `poll_one` function runs it and returns 1
- Otherwise, the function returns immediately and returns 0

Operation pending, ready to be ran without blocking, usually means any of:

- A timer that has expired, and its `async_wait` handler needs to be called
- An I/O operation that has completed (such as, `async_read`), and its handler needs to be called
- A custom function handler that was previously added to `io_services` instance's queue (this is explained in detail in the following section)

You can use `poll_one` to make sure all handlers of completed I/O operations ran, and then do some other coding as well:

```
io_service service;
while ( true) {
    // run all handlers of completed IO operations
    while ( service.poll_one() ) ;
    // ... do other work here ...
}
```

The `poll()` function will run all operations that are pending and can be run without blocking. The following code is equivalent:

```
io_service service;
service.poll(); // OR
while ( service.poll_one() ) ;
```

All the preceding functions will throw a `boost::system::system_error` exception on failure. This should never happen; an error thrown here is usually fatal, maybe an out-of-resources error or so, or maybe one of your handlers threw an exception. Anyway, each of the functions has an overload that does not throw and takes a `boost::system::error_code` argument and is set upon its return:

```
io_service service;
boost::system::error_code err = 0;
service.run(err);
if ( err ) std::cout << "Error " << err << std::endl;
```

Asynchronous work

Asynchronous work is not just about asynchronously accepting clients connecting to a server, asynchronous reads from or writes to sockets. It encompasses any operation that can execute asynchronously.

By default, you don't know the order in which each asynchronous handler function is called. Besides, the usual asynchronous calls (coming from asynchronous socket reads/writes/accepts). You can use `service.post()` to post your custom function to be called asynchronously. For instance:

```
#include <boost/thread.hpp>
#include <boost/bind.hpp>
#include <boost/asio.hpp>
#include <iostream>
using namespace boost::asio;
io_service service;
void func(int i) {
    std::cout << "func called, i= " << i << std::endl;
}
void worker_thread() {
    service.run();
}
```

```
int main(int argc, char* argv[]) {
    for ( int i = 0; i < 10; ++i)
        service.post(boost::bind(func, i));
    boost::thread_group threads;
    for ( int i = 0; i < 3; ++i)
        threads.create_thread(worker_thread);
    // wait for all threads to be created
    boost::this_thread::sleep( boost::posix_time::millisec(500));
    threads.join_all();
}
```

In the preceding example, `service.post(some_function)` adds an asynchronous function call. This function returns immediately, after requesting the `io_service` instance to invoke the given `some_function`, in one of the threads that called `service.run()`. In our case, this is one of the three threads we previously created. You can't be certain about the order of the asynchronous function calls. You should not expect them to be called in the order they were posted (`post()`). A possible outcome of running the previous example is as follows:

```
func called, i= 0
func called, i= 2
func called, i= 1
func called, i= 4
func called, i= 3
func called, i= 6
func called, i= 7
func called, i= 8
func called, i= 5
func called, i= 9
```

There will be times when you want to order some of the asynchronous handler functions. Say, you have to go to restaurant (`go_to_restaurant`), order (`order`), and eat (`eat`). You'll want to go to the restaurant first, then order, and finally eat. For this, you'll use `io_service::strand`, which will order your asynchronous handler calls. Consider the following example:

```
using namespace boost::asio;
io_service service;
void func(int i) {
    std::cout << "func called, i= " << i << "/"
               << boost::this_thread::get_id() << std::endl;
}
```

```
void worker_thread() {
    service.run();
}

int main(int argc, char* argv[])
{
    io_service::strand strand_one(service), strand_two(service);
    for ( int i = 0; i < 5; ++i)
        service.post( strand_one.wrap( boost::bind(func, i)));
    for ( int i = 5; i < 10; ++i)
        service.post( strand_two.wrap( boost::bind(func, i)));
    boost::thread_group threads;
    for ( int i = 0; i < 3; ++i)
        threads.create_thread(worker_thread);
    // wait for all threads to be created
    boost::this_thread::sleep( boost::posix_time::millisec(500));
    threads.join_all();
}
```

In the preceding code, we made sure that the first five and the last five were serialized namely, func called, i = 0 is called before func called, i = 1, which is called before func called, i = 2, and so on. The same goes for func called, i = 5, which is called before func called, i = 6, and func called, i = 6 is called before func called, i = 7, and so on. You should note that even if the function calls are serialized, that does not mean they will all happen in the same thread. A possible outcome of running this program can be:

```
func called, i= 0/002A60C8
func called, i= 5/002A6138
func called, i= 6/002A6530
func called, i= 1/002A6138
func called, i= 7/002A6530
func called, i= 2/002A6138
func called, i= 8/002A6530
func called, i= 3/002A6138
func called, i= 9/002A6530
func called, i= 4/002A6138
```

Asynchronous `post()` versus `dispatch()` versus `wrap()`

Boost.Asio provides three ways to add your function handler to be asynchronously called:

- `service.post(handler)`: This function guarantees that it returns immediately after it has requested the `io_service` instance to invoke the given function handler. The handler will be called later in one of the threads that has called `service.run()`.
- `service.dispatch(handler)`: This requests the `io_service` instance to invoke the given function handler, but in addition, it can execute the handler inside the function if the current thread has called `service.run()`.
- `service.wrap(handler)`: This function creates a wrapper function that when called will call `service.dispatch(handler)`. This is a bit confusing; I'll explain shortly what this means.

You've seen an example of `service.post()` in the previous section, together with a possible outcome of running the program. Lets modify it, and see how `service.dispatch()` affects the outcome:

```
using namespace boost::asio;
io_service service;
void func(int i) {
    std::cout << "func called, i= " << i << std::endl;
}
void run_dispatch_and_post() {
    for ( int i = 0; i < 10; i += 2) {
        service.dispatch(boost::bind(func, i));
        service.post(boost::bind(func, i + 1));
    }
}
int main(int argc, char* argv[]) {
    service.post(run_dispatch_and_post);
    service.run();
}
```

Before explaining what's happening, let's see the results by running the program:

```
func called, i= 0
func called, i= 2
func called, i= 4
func called, i= 6
func called, i= 8
func called, i= 1
func called, i= 3
func called, i= 5
func called, i= 7
func called, i= 9
```

Even numbers are written first, then the odd ones. This is because I use `dispatch()` to write the even numbers, and `post()` to write the odd numbers. `dispatch()` will call the handler before it returns, because the current thread has called `service.run()`, while `post()` always returns immediately.

Now, let's talk about `service.wrap(handler)`. The `wrap()` returns a functor, which can be further used as an argument to another functions:

```
using namespace boost::asio;
io_service service;
void dispatched_func_1() {
    std::cout << "dispatched 1" << std::endl;
}
void dispatched_func_2() {
    std::cout << "dispatched 2" << std::endl;
}
void test(boost::function<void()> func) {
    std::cout << "test" << std::endl;
    service.dispatch(dispatched_func_1);
    func();
}
void service_run() {
    service.run();
}
int main(int argc, char* argv[]) {
    test( service.wrap(dispatched_func_2));
    boost::thread th(service_run);
    boost::this_thread::sleep( boost::posix_time::millisec(500));
    th.join();
}
```

The line `test(service.wrap(dispatched_func_2));` will wrap `dispatched_func_2` and create a functor passed as an argument to `test`. When `test()` is called, it will dispatch calling function 1, and call `func()`. At this point, you'll see that calling `func()` is equivalent to `service.dispatch(dispatched_func_2)`, because they are called sequentially. The output of the program confirms it:

```
test
dispatched 1
dispatched 2
```

The `io_service::strand` class (used for serializing asynchronous actions) also contains the member functions `poll()`, `dispatch()` and `wrap()`. Their meaning is the same as `io_service`'s `poll()`, `dispatch()`, and `wrap()`. However, most of the time you will only use `io_service::strand::wrap()` function as argument to `io_service::poll()` or `io_service::dispatch()`.

Staying alive

Say, you do the following operation:

```
io_service service;
ip::tcp::socket sock(service);
char buff[512];
...
read(sock, buffer(buff));
```

In this case, `sock` and `buff` must both outlive the call to `read()`. In other words, they must be valid after the call to `read()` returns. This is just what you'd expect; all arguments you pass to a function should be valid inside the function. Things get more complicated when we go asynchronous:

```
io_service service;
ip::tcp::socket sock(service);
char buff[512];
void on_read(const boost::system::error_code &, size_t) {}
...
async_read(sock, buffer(buff), on_read);
```

In this case, `sock` and `buff` must outlive the `read` operation itself, which we don't know when will happen, since it's asynchronous.

When using socket buffers, you can have a buffer instance outlive an asynchronous call (make use of `boost::shared_array<>`). We can use the same principle here by creating a class that internally holds the socket and its read/write buffers. Then, for all asynchronous calls, I will pass a `boost::bind` functor with a shared pointer:

```
using namespace boost::asio;
io_service service;
struct connection : boost::enable_shared_from_this<connection> {
    typedef boost::system::error_code error_code;
    typedef boost::shared_ptr<connection> ptr;
    connection() : sock_(service), started_(true) {}
    void start(ip::tcp::endpoint ep) {
        sock_.async_connect(ep,
            boost::bind(&connection::on_connect, shared_from_this(),
                _1));
    }
    void stop() {
        if ( !started_ ) return;
        started_ = false;
        sock_.close();
    }
    bool started() { return started_; }
private:
    void on_connect(const error_code & err) {
        // here you decide what to do with the connection: read or
        write
        if ( !err ) do_read();
        else stop();
    }
    void on_read(const error_code & err, size_t bytes) {
        if ( !started() ) return;
        std::string msg(read_buffer_, bytes);
        if ( msg == "can_login" ) do_write("access_data");
        else if ( msg.find("data ") == 0 ) process_data(msg);
        else if ( msg == "login_fail" ) stop();
    }
    void on_write(const error_code & err, size_t bytes) {
        do_read();
    }
    void do_read() {
```



```
        sock_.async_read_some(buffer(read_buffer_),
                               boost::bind(&connection::on_read, shared_from_this(),
                                             _1, _2));
    }
    void do_write(const std::string & msg) {
        if ( !started() ) return;
        // note: in case you want to send several messages before
        //         doing another async_read, you'll need several write
        buffers!
        std::copy(msg.begin(), msg.end(), write_buffer_);
        sock_.async_write_some(buffer(write_buffer_, msg.size()),
                               boost::bind(&connection::on_write, shared_from_this(),
                                             _1, _2));
    }
    void process_data(const std::string & msg) {
        // process what comes from server, and then perform another
        write
    }
private:
    ip::tcp::socket sock_;
    enum { max_msg = 1024 };
    char read_buffer_[max_msg];
    char write_buffer_[max_msg];
    bool started_;
};

int main(int argc, char* argv[]) {
    ip::tcp::endpoint ep( ip::address::from_string("127.0.0.1"),
                          8001);
    connection::ptr(new connection)->start(ep);
}
```

In all asynchronous calls, we pass a `boost::bind` functor as an argument. That functor internally keeps a shared pointer to the connection instance. As long as there's an asynchronous operation pending, Boost.Asio will keep a copy of the `boost::bind` functor, which keeps a shared pointer to the connection instance, keeping the connection instance alive. Problem solved!

Of course, the connection class is just a skeleton class; you'll have to adapt it to your needs (it will look quite different in the case of a server).

Notice how easy you create a new connection, `connection::ptr(new connection)->start(ep)`. This starts the (asynchronous) connection to the server. When you want to close this connection, you'll call `stop()`.

Once the instance is started (`start()`), it will wait to get connected. When connection happens, `on_connect()` is called. If there's no error, it performs a read operation (`do_read()`). Once the read operation completes, you interpret the message; your application's `on_read()` will look very different. When you write a message, you have to copy it to the buffer, and then send it just like I did in `do_write()`, because again, the buffer needs to outlive the asynchronous write. One final note – when writing, remember that you have to specify how much to write, otherwise, the whole buffer will be sent.

Summary

The Networking API is rather vast. This chapter was implemented as a reference, which you should come back to, while implementing your own networking applications.

Boost.Asio implemented the concept of endpoints, which you can think of as an IP and a port. If you don't know the exact IP, you can use a `resolver` object to turn a hostname, such as `www.yahoo.com` into one or several IPs.

We've also seen the `socket` classes, which are at the core of the API. Boost.Asio provides implementations for TCP, UDP, and ICMP, but you can extend it with your own protocols; it's not a job for the faint-hearted, though.

Asynchronous programming is a necessary evil. You've seen why you sometimes need it, especially when writing servers. Usually, you'll be happy with calling `service.run()` to implement the asynchronous loop, but just in case you need to go advanced, you can use `run_one()`, `poll()`, or `poll_one()`.

When going asynchronous, you can also have your own functions executed asynchronously; just use `service.post()` or `service.dispatch()`.

Finally, in order for both the `socket` and the `buffer` (`read` or `write`) to be alive for the whole period of the asynchronous operation (until it completes), we need to take special precautions. Your `connection` class should derive from `enabled_shared_from_this`, keep internally all its needed buffers, and each asynchronous call will pass a shared pointer to the `this` operation.

The next chapter will really put you to work; lots of hands-on coding while implementing echo client/server applications.

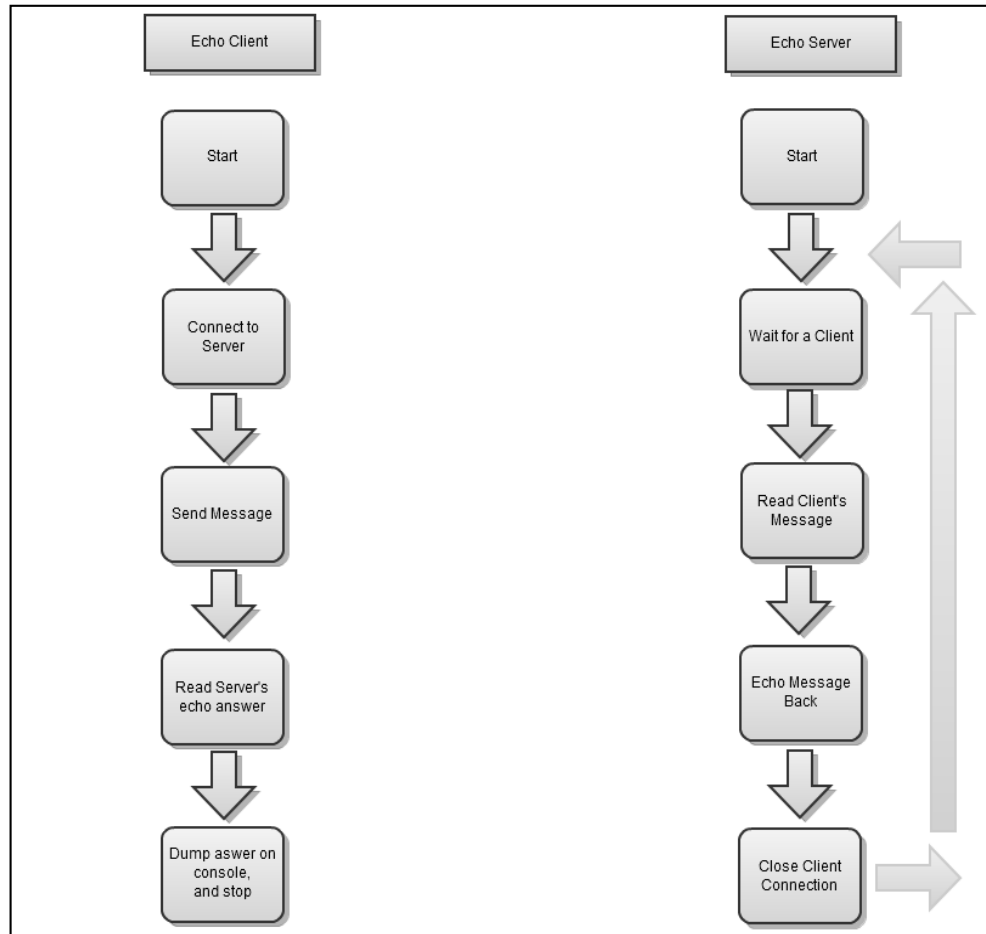
3

Echo Server/Clients

In this chapter, we'll implement a small client/server application, which is probably the easiest client/server application you will ever write. This is the Echo application, a server that echoes back anything a client writes, and then closes the client's connection. The server can handle any number of clients. As each client connects, it sends a message. The server receives the full message and sends it back. After that, it closes the connection.

Therefore, each Echo client connects to the server, sends a message, and reads what the server replies, making sure it's the same message it sent and finishes talking to the server.

We will implement first a synchronous application, and then an asynchronous application, so you can easily compare them:



Some of the following code has been trimmed to save space. You will find the full code in the code accompanying this book.

TCP Echo server/clients

For TCP, we can have an extra guarantee; each message ends in line feed ('`\n`'). Coding Echo servers/clients synchronously is extremely easy.

We will present programs, such as synchronous client, a synchronous server, a asynchronous client, and an asynchronous server.

TCP synchronous client

In most non-trivial examples, it's usually the client that is easier to code, than the server (since the server needs to deal with multiple clients).

The following code shows an exception to the rule:

```
ip::tcp::endpoint ep( ip::address::from_string("127.0.0.1"), 8001);

size_t read_complete(char * buf, const error_code & err, size_t bytes)
{
    if ( err) return 0;
    bool found = std::find(buf, buf + bytes, '\n') < buf + bytes;
    // we read one-by-one until we get to enter, no buffering
    return found ? 0 : 1;
}

void sync_echo(std::string msg) {
    msg += "\n";
    ip::tcp::socket sock(service);
    sock.connect(ep);
    sock.write_some(buffer(msg));
    char buf[1024];
    int bytes = read(sock, buffer(buf), boost::bind(read_
complete,buf,_1,_2));
    std::string copy(buf, bytes - 1);
    msg = msg.substr(0, msg.size() - 1);
    std::cout << "server echoed our " << msg << ": "
                << (copy == msg ? "OK" : "FAIL") << std::endl;
    sock.close();
}

int main(int argc, char* argv[]) {
    char* messages[] = { "John says hi", "so does James",
                        "Lucy just got home", "Boost.Asio is Fun!", 0
    };

    boost::thread_group threads;
    for ( char ** message = messages; *message; ++message) {
        threads.create_thread( boost::bind(sync_echo, *message));
        boost::this_thread::sleep( boost::posix_time::millisec(100));
    }
    threads.join_all();
}
```

The function to watch for is `sync_echo`. It contains all the logic for connecting to a server, sending it a message and waiting for the echo back.

You'll notice that, for reading, I've used the free function `read()`, because I want to read everything up to `'\n'`. The `sock.read_some()` function would not be enough, since that would only read what's available, which is not necessarily the whole message.

The third argument to the `read()` function is a completion handler. It will return 0 when it's read the full message. Otherwise, it will return the maximum buffer it can read in the next step (until read is complete). In our case, this is always 1, because we never want to mistakenly read more than we need.

In `main()`, we create several threads; one thread for each message to send to the client, and wait for them to complete. If you run the program, you'll see the following output:

```
server echoed our John says hi: OK
server echoed our so does James: OK
server echoed our Lucy just got home: OK
server echoed our Boost.Asio is Fun!: OK
```

Notice that since we're synchronous, there's no need to call `service.run()`.

TCP synchronous server

The Echo synchronous server is quite easy to write, as shown in the following code snippet:

```
io_service service;
size_t read_complete(char * buff, const error_code & err, size_t
bytes) {
    if ( err) return 0;
    bool found = std::find(buff, buff + bytes, '\n') < buff + bytes;
    // we read one-by-one until we get to enter, no buffering
    return found ? 0 : 1;
}
void handle_connections() {
    ip::tcp::acceptor acceptor(service, ip::tcp::endpoint(ip::tcp::
:v4(), 8001));
    char buff[1024];
    while ( true) {
        ip::tcp::socket sock(service);
        acceptor.accept(sock);
        int bytes = read(sock, buffer(buff),
```

```

        boost::bind(read_complete, buff, _1, _2));
        std::string msg(buff, bytes);
        sock.write_some(buffer(msg));
        sock.close();
    }
}

int main(int argc, char* argv[]) {
    handle_connections();
}

```

The logic of the server is `handle_connections()`. Since we're single-threaded, we accept a new client, read the message it sends us, echo it back, and then wait for the next client. Let's say, if two clients connect at once, the second client will have to wait for the server to service the first client.

Notice again that since we're synchronous, there's no need to call `service.run()`.

TCP asynchronous client

Once we go asynchronous, the code becomes a bit more complicated. We'll model the connection class shown in *Chapter 2, Staying Alive*.

By looking at the following code snippets in this section, you will notice that every asynchronous operation starts a new asynchronous operation, keeping the `service.run()` busy.

First, the core functionality is:

```

#define MEM_FN(x)      boost::bind(&self_type::x, shared_from_this())
#define MEM_FN1(x,y)   boost::bind(&self_type::x, shared_from_
this(), y)
#define MEM_FN2(x,y,z) boost::bind(&self_type::x, shared_from_
this(), y, z)

class talk_to_svr : public boost::enable_shared_from_this<talk_to_svr>
, boost::noncopyable {
    typedef talk_to_svr self_type;
    talk_to_svr(const std::string & message)
        : sock_(service), started_(true), message_(message) {}
    void start(ip::tcp::endpoint ep) {
        sock_.async_connect(ep, MEM_FN1(on_connect, _1));
    }
public:

```



```
typedef boost::system::error_code error_code;
typedef boost::shared_ptr<talk_to_svr> ptr;
static ptr start(ip::tcp::endpoint ep, const std::string &
message) {
    ptr new_(new talk_to_svr(message));
    new_->start(ep);
    return new_;
}
void stop() {
    if ( !started_) return;
    started_ = false;
    sock_.close();
}
bool started() { return started_; }
...
private:
    ip::tcp::socket sock_;
    enum { max_msg = 1024 };
    char read_buffer_[max_msg];
    char write_buffer_[max_msg];
    bool started_;
    std::string message_;
};
```

We want to always use shared pointers to `talk_to_svr`, so that as long as there are asynchronous operations on an instance of `talk_to_svr`, that instance is alive. In order to avoid mistakes, such as constructing an instance of the `talk_to_svr` object on the stack, I've made the constructor private and disallowed copy construction (derived from `boost::noncopyable`).

We have the core functions, such as `start()`, `stop()`, and `started()` that do just what their names say. To construct a connection, just call `talk_to_svr::start(endpoint, message)`. We also have one read and one write buffer (`read_buffer_` and `write_buffer_`).

The `MEM_FN*` macros are convenience macros, and they enforce always using a shared pointer to `*this`, via the `shared_ptr_from_this()` function.

The following lines are very different than explained earlier:

```
// equivalent to "sock_.async_connect(ep, MEM_FN1(on_connect, _1));"
sock_.async_connect(ep,
    boost::bind(&talk_to_svr::on_connect, shared_ptr_from_this(), _1));
sock_.async_connect(
    ep, boost::bind(&talk_to_svr::on_connect, this, _1));
```

In the former case, we're creating the `async_connect` completion handler correctly; it will hold a shared pointer to the `talk_to_server` instance until it calls the completion handler, thus, making sure we're still alive when that happens.

In the latter case, we're creating the completion handler incorrectly. By the time it gets called, the `talk_to_server` instance could have been deleted!

To read from or write to the socket, you'll use following code snippet:

```
void do_read() {
    async_read(sock_, buffer(read_buffer_),
               MEM_FN2(read_complete, _1, _2), MEM_FN2(on_read, _1, _2));
}
void do_write(const std::string & msg) {
    if ( !started() ) return;
    std::copy(msg.begin(), msg.end(), write_buffer_);
    sock_.async_write_some( buffer(write_buffer_, msg.size()),
                           MEM_FN2(on_write, _1, _2));
}
size_t read_complete(const boost::system::error_code & err, size_t
bytes) {
    // similar to the one shown in TCP Synchronous Client
}
```

The `do_read()` function will make sure that we read a line from the server, at which point `on_read()` is called. The `do_write()` function will first copy the message into the buffer (since `msg` will probably go out of scope and be destroyed by the time the `async_write` actually takes place), and then make sure `on_write()` is called after the actual write takes place.

And the most important functions, the one that contain the main logic of the class:

```
void on_connect(const error_code & err) {
    if ( !err)      do_write(message_ + "\n");
    else           stop();
}
void on_read(const error_code & err, size_t bytes) {
    if ( !err) {
        std::string copy(read_buffer_, bytes - 1);
        std::cout << "server echoed our " << message_ << ": "
                   << (copy == message_ ? "OK" : "FAIL") <<
        std::endl;
    }
    stop();
}
```

```
void on_write(const error_code & err, size_t bytes) {
    do_read();
}
```

After we're connected, we send the message to the server, `do_write()`. When the write operation is finished, `on_write()` gets called, which initiates a `do_read()` function. When `do_read()` is complete, `on_read()` gets called; here, we simply check that the message from the server is simply an echo, and exit from it.

We'll send three messages to the server just to make it a bit more interesting:

```
int main(int argc, char* argv[]) {
    ip::tcp::endpoint ep( ip::address::from_string("127.0.0.1"),
    8001);
    char* messages[] = { "John says hi", "so does James", "Lucy got
    home", 0 };
    for ( char ** message = messages; *message; ++message) {
        talk_to_srv::start( ep, *message);
        boost::this_thread::sleep( boost::posix_time::millisec(100));
    }
    service.run();
}
```

The preceding code snippet will generate the following code:

```
server echoed our John says hi: OK
server echoed our so does James: OK
server echoed our Lucy just got home: OK
```

TCP asynchronous server

The core functionality is similar to the one from the asynchronous client, shown as follows:

```
class talk_to_client : public boost::enable_shared_from_this<talk_to_
client>
    , boost::noncopyable {
    typedef talk_to_client self_type;
    talk_to_client() : sock_(service), started_(false) {}
public:
    typedef boost::system::error_code error_code;
    typedef boost::shared_ptr<talk_to_client> ptr;
```

```

    void start() {
        started_ = true;
        do_read();
    }
    static ptr new_() {
        ptr new_(new talk_to_client);
        return new_;
    }
    void stop() {
        if ( !started_ ) return;
        started_ = false;
        sock_.close();
    }
    ip::tcp::socket & sock() { return sock_; }
    ...
private:
    ip::tcp::socket sock_;
    enum { max_msg = 1024 };
    char read_buffer_[max_msg];
    char write_buffer_[max_msg];
    bool started_;
};

```

Since we've a very simple Echo server, there is no need for an `is_started()` function. For each client, just read its message, echo it back, and close it.

The `do_read()`, `do_write()` and `read_complete()` functions are exactly the same as in the TCP asynchronous client.

The main logic of the class is again in `on_read()` and `on_write()`:

```

    void on_read(const error_code & err, size_t bytes) {
        if ( !err ) {
            std::string msg(read_buffer_, bytes);
            do_write(msg + "\n");
        }
        stop();
    }
    void on_write(const error_code & err, size_t bytes) {
        do_read();
    }

```

Dealing with the clients is done as follows:

```
ip::tcp::acceptor acceptor(service, ip::tcp::endpoint(ip::tcp::v4(),
8001));
void handle_accept(talk_to_client::ptr client, const error_code & err)
{
    client->start();
    talk_to_client::ptr new_client = talk_to_client::new_();
    acceptor.async_accept(new_client->sock(),
        boost::bind(handle_accept, new_client, _1));
}
int main(int argc, char* argv[]) {
    talk_to_client::ptr client = talk_to_client::new_();
    acceptor.async_accept(client->sock(),
        boost::bind(handle_accept, client, _1));
    service.run();
}
```

Each time a client connects to the server, `handle_accept` is called, which will asynchronously start reading from that client, and also asynchronously wait for a new client.

The code

You'll find all four applications (TCP Echo Sync Client, TCP Echo Sync Server, TCP Echo Async Client, TCP Echo Async Server) in the code accompanying this book. When testing, you can use any client/server combination (such as, an asynchronous client versus a synchronous server).

UDP Echo server/clients

Since in UDP not all messages reach the recipient, we can't have the "message ends in enter" guarantee.

Each message we receive, we simply echo back with no socket to close (on the server side), since we're UDP.

UDP synchronous Echo client

The UDP Echo client is simpler than the TCP Echo client:

```
ip::udp::endpoint ep( ip::address::from_string("127.0.0.1"), 8001);
void sync_echo(std::string msg) {
    ip::udp::socket sock(service, ip::udp::endpoint(ip::udp::v4(), 0)
);
    sock.send_to(buffer(msg), ep);
    char buff[1024];
    ip::udp::endpoint sender_ep;
    int bytes = sock.receive_from(buffer(buff), sender_ep);
    std::string copy(buff, bytes);
    std::cout << "server echoed our " << msg << ": "
                << (copy == msg ? "OK" : "FAIL") << std::endl;
    sock.close();
}
int main(int argc, char* argv[]) {
    char* messages[] = { "John says hi", "so does James", "Lucy got
home", 0 };
    boost::thread_group threads;
    for ( char ** message = messages; *message; ++message) {
        threads.create_thread( boost::bind(sync_echo, *message));
        boost::this_thread::sleep( boost::posix_time::millisec(100));
    }
    threads.join_all();
}
```

The whole logic is in `sync_echo()`; connect to the server, send the message, receive the echo from server, and close the connection.

UDP synchronous Echo server

The UDP Echo server is the easiest server you'll ever write:

```
io_service service;
void handle_connections() {
    char buff[1024];
    ip::udp::socket sock(service, ip::udp::endpoint(ip::udp::v4(),
8001));
    while ( true) {
        ip::udp::endpoint sender_ep;
        int bytes = sock.receive_from(buffer(buff), sender_ep);
        std::string msg(buff, bytes);
        sock.send_to(buffer(msg), sender_ep);
    }
}
int main(int argc, char* argv[]) {
    handle_connections();
}
```

That's simple, and quite self-explanatory.

I'll leave the asynchronous UDP client and server as an exercise for the reader.

Summary

We've written full applications and finally put Boost.Asio to work. The Echo application is a very good tool to start learning a library. You can always study and run the code shown in this chapter to easily remember the library's fundamentals.

In the following chapter, we'll build more complex client/server applications, making sure we avoid pitfalls, such as memory leaks, deadlocks, and so on.

4

Client and Server

In this chapter, we're about to delve into building non-trivial client and server applications using Boost.Asio. You can run and test them, and once you understand them, you can use them as skeletons to build your own applications.

In the following examples:

- The client logs in to the server with a username (no password)
- All connections are initiated by the client, where client asks and server answers
- All requests and answers are finished with a line feed (`'\n'`)
- Server disconnects any client that hasn't pinged for 5 seconds

The client can make the following requests:

- Get a list of all connected clients
- The client can ping, and when it pings, the server answers either with `ping ok` or `ping client_list_changed` (in the latter case, the client re-requests the list of connected clients)

To keep things interesting, we'll add a few twists:

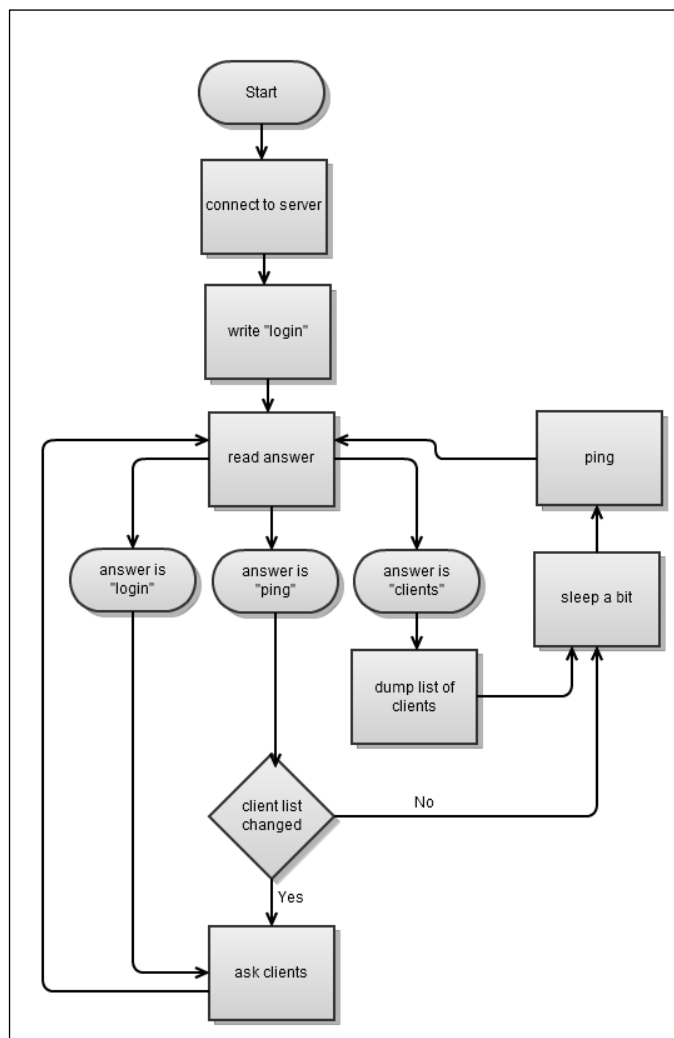
- Each client application logs in six with user connections, such as John, James, Lucy, Tracy, Frank, Abby
- Each client connection pings the server at random times (random of 7 seconds; thus, every now and then, the server will drop a connection)

The synchronous client/server

First, we'll implement the synchronous application. You'll see that the code is straightforward and easy to read. However, the networking part does need its own thread(s), since all networking calls are blocking.

Synchronous client

The synchronous client does exactly what you'd expect in a serial fashion; connects to the server, logs in to the server, then performs the connection loop, such as sleep a bit, make a request, read the server's answer, then sleep a bit again, and so on.



Since we're synchronous, let's keep things simple. First, connect to the server, then the loop, as follows:

```
ip::tcp::endpoint ep( ip::address::from_string("127.0.0.1"), 8001);
void run_client(const std::string & client_name) {
    talk_to_svr client(client_name);
    try {
        client.connect(ep);
        client.loop();
    } catch(boost::system::system_error & err) {
        std::cout << "client terminated " << std::endl;
    }
}
```

The following code snippet shows the `talk_to_svr` class:

```
struct talk_to_svr {
    talk_to_svr(const std::string & username)
        : sock_(service), started_(true), username_(username) {}
    void connect(ip::tcp::endpoint ep) {
        sock_.connect(ep);
    }
    void loop() {
        write("login " + username_ + "\n");
        read_answer();
        while ( started_ ) {
            write_request();
            read_answer();
            boost::this_thread::sleep(millisec(rand() % 7000));
        }
    }
    std::string username() const { return username_; }
    ...
private:
    ip::tcp::socket sock_;
    enum { max_msg = 1024 };
    int already_read_;
    char buff_[max_msg];
    bool started_;
    std::string username_;
};
```

In the loop, we just put a bit and a ping to sleep, and read the server's answer. We put it to sleep at random (sometimes over 5 seconds), so that the server will disconnect us at some point:

```
void write_request() {
    write("ping\n");
}
void read_answer() {
    already_read_ = 0;
    read(sock_, buffer(buff_),
        boost::bind(&talk_to_svr::read_complete, this, _1, _2));
    process_msg();
}
void process_msg() {
    std::string msg(buff_, already_read_);
    if ( msg.find("login ") == 0) on_login();
    else if ( msg.find("ping") == 0) on_ping(msg);
    else if ( msg.find("clients ") == 0) on_clients(msg);
    else std::cerr << "invalid msg " << msg << std::endl;
}
```

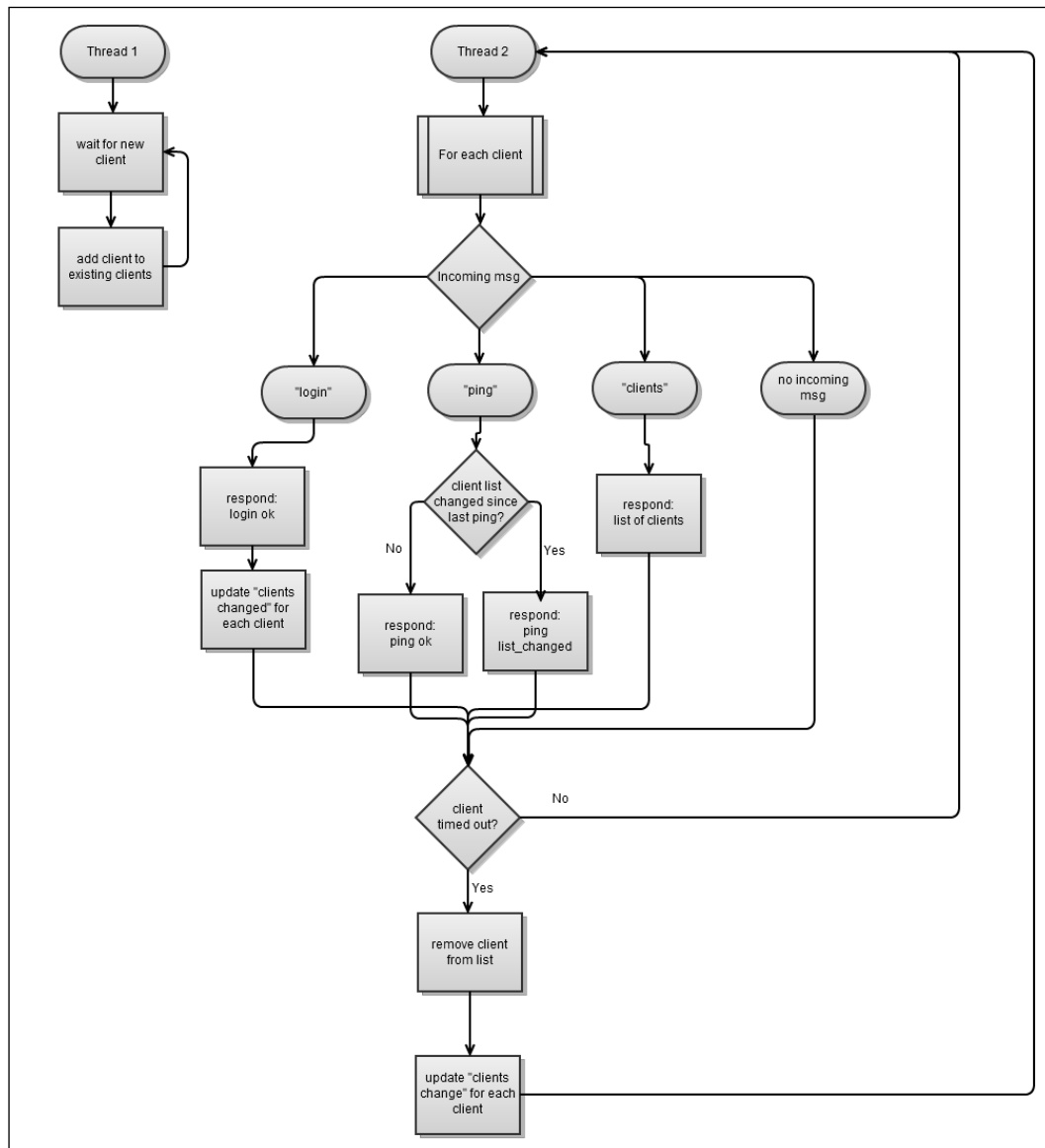
For reading the answer, we use `read_complete`, explained extensively in the previous chapter, to make sure that we read up to the line feed (`'\n'`). The logic is in `process_msg()`, where we read the client's answer, and dispatch to the right function:

```
void on_login() { do_ask_clients(); }
void on_ping(const std::string & msg) {
    std::istringstream in(msg);
    std::string answer;
    in >> answer >> answer;
    if ( answer == "client_list_changed")
        do_ask_clients();
}
void on_clients(const std::string & msg) {
    std::string clients = msg.substr(8);
    std::cout << username_ << ", new client list:" << clients;
}
void do_ask_clients() {
    write("ask_clients\n");
    read_answer();
}
void write(const std::string & msg) { sock_.write_some(buffer(msg)); }
size_t read_complete(const boost::system::error_code & err, size_t
bytes) {
    // ... same as before
}
```

When reading the server's answer to our ping, if we get `client_list_changed`, we ask again for the list of clients.

Synchronous server

The synchronous server is quite simple as well. It needs two threads, one for listening to new clients and one for processing existing clients. It cannot use a single thread; waiting for a new client is a blocking operation, thus, we need an extra thread to handle the existing clients.



The server, as expected, is harder to write than the client. For one thing, it needs to manage all connected clients. Since we're synchronous, we'll need at least two threads, one that accepts new clients (since `accept()` is blocking) and one that answers existing clients:

```
void accept_thread() {
    ip::tcp::acceptor acceptor(service,
                               ip::tcp::endpoint(ip::tcp::v4(),
8001));
    while ( true) {
        client_ptr new_( new talk_to_client);
        acceptor.accept(new_>sock());
        boost::recursive_mutex::scoped_lock lk(cs);
        clients.push_back(new_);
    }
}

void handle_clients_thread() {
    while ( true) {
        boost::this_thread::sleep( millisec(1));
        boost::recursive_mutex::scoped_lock lk(cs);
        for(array::iterator b = clients.begin(), e = clients.end(); b
!= e; ++b)
            (*b)->answer_to_client();
        // erase clients that timed out
        clients.erase(std::remove_if(clients.begin(), clients.end(),
            boost::bind(&talk_to_client::timed_out, _1)),
clients.end());
    }
}

int main(int argc, char* argv[]) {
    boost::thread_group threads;
    threads.create_thread(accept_thread);
    threads.create_thread(handle_clients_thread);
    threads.join_all();
}
```

We need to keep a list of all clients in order to process incoming requests from each of them.

Each `talk_to_client` instance holds a socket. The socket class is not copy-constructible, thus, if you want to hold it into an `std::vector` function, you need to hold a shared pointer to it. There are two ways to go about this: either inside `talk_to_client`, hold a shared pointer to a socket and then have an array of `talk_to_client` instances, or have the `talk_to_client` instance hold a the socket by value, and have an array of shared pointers to `talk_to_client`. I chose the latter, but you can go either way:

```
typedef boost::shared_ptr<talk_to_client> client_ptr;
typedef std::vector<client_ptr> array;
array clients;
boost::recursive_mutex cs; // thread-safe access to clients array
```

The main `talk_to_client` code is as follows:

```
struct talk_to_client : boost::enable_shared_from_this<talk_to_client>
{
    talk_to_client() { ... }
    std::string username() const { return username_; }
    void answer_to_client() {
        try {
            read_request();
            process_request();
        } catch ( boost::system::system_error& ) {
            stop();
        }
        if ( timed_out() )
            stop();
    }
    void set_clients_changed() { clients_changed_ = true; }
    ip::tcp::socket & sock() { return sock_; }
    bool timed_out() const {
        ptime now = microsec_clock::local_time();
        long long ms = (now - last_ping).total_milliseconds();
        return ms > 5000 ;
    }
    void stop() {
        boost::system::error_code err; sock_.close(err);
    }
}
```

```
void read_request() {
    if ( sock_.available())
        already_read_ += sock_.read_some(
            buffer(buff_ + already_read_, max_msg - already_
read_));
}
...
private:
    // ... same as in Synchronous Client
    bool clients_changed_;
    ptime last_ping;
};
```

The preceding code is pretty self-explanatory. The most important function is `read_request()`. This will read only if there's data available, thus, the server never gets blocked:

```
void process_request() {
    bool found_enter = std::find(buff_, buff_ + already_read_, '\n')
        < buff_ + already_read_;
    if ( !found_enter)
        return; // message is not full
    // process the msg
    last_ping = microsec_clock::local_time();
    size_t pos = std::find(buff_, buff_ + already_read_, '\n') -
buff_;
    std::string msg(buff_, pos);
    std::copy(buff_ + already_read_, buff_ + max_msg, buff_);
    already_read_ -= pos + 1;

    if ( msg.find("login ") == 0) on_login(msg);
    else if ( msg.find("ping") == 0) on_ping();
    else if ( msg.find("ask_clients") == 0) on_clients();
    else std::cerr << "invalid msg " << msg << std::endl;
}
void on_login(const std::string & msg) {
    std::istringstream in(msg);
    in >> username_ >> username_;
    write("login ok\n");
    update_clients_changed();
}
```

```

void on_ping() {
    write(clients_changed_ ? "ping client_list_changed\n" : "ping
ok\n");
    clients_changed_ = false;
}
void on_clients() {
    std::string msg;
    { boost::recursive_mutex::scoped_lock lk(cs);
      for( array::const_iterator b = clients.begin(), e = clients.
end() ;
          b != e; ++b)
        msg += (*b)->username() + " ";
    }
    write("clients " + msg + "\n");
}
void write(const std::string & msg) { sock_.write_some(buffer(msg)); }

```

Take a look at `process_request()`. After we've read only as much as was available, we need to know if we read the full message (if `found_enter` is true). If so, we're protecting ourselves against maybe reading more than one message (anything after `'\n'` is saved in the buffer), and then we interpret the fully read message. The rest of the code is straightforward.

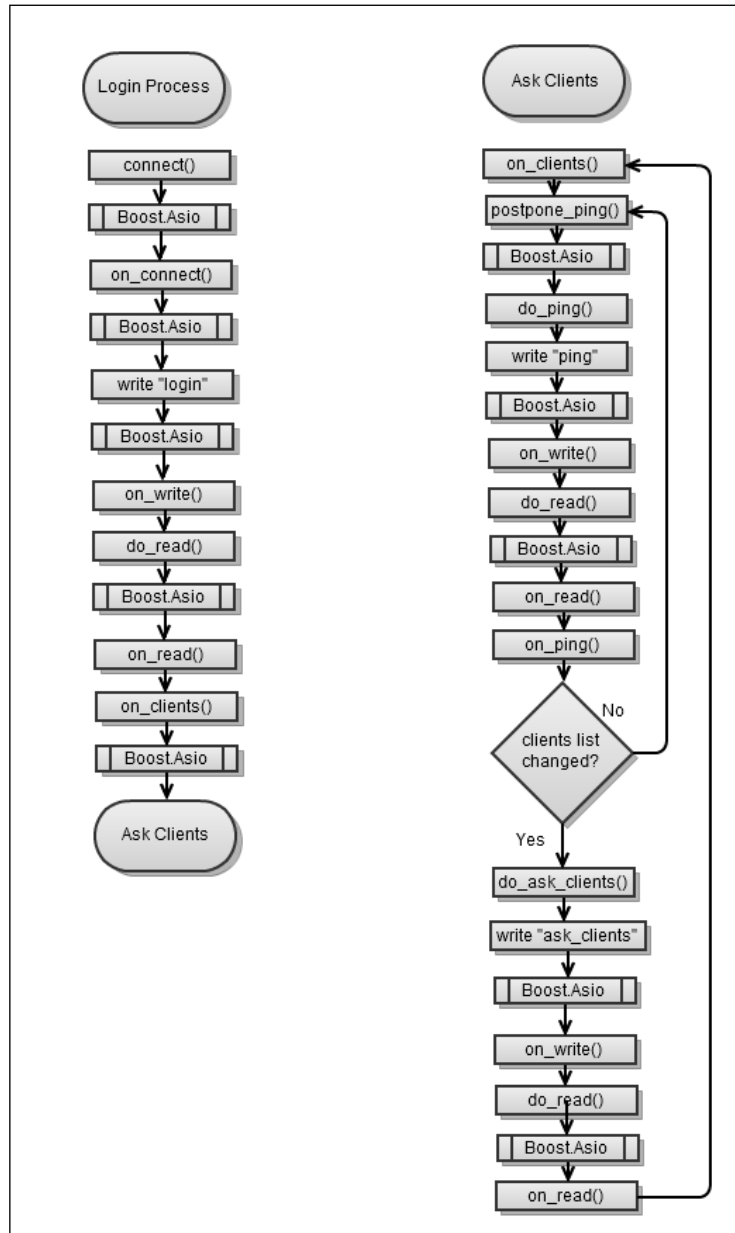
The asynchronous client/server

And now, for the fun (and hard) part, let's go asynchronous!

When checking out the diagrams, always understand that Boost.Asio means an asynchronous call performed by Boost.Asio. For instance, `do_read()`, `Boost.Asio`, and `on_read()` indicates the logical flow from `do_read()` to `on_read()`, but you will never know the time it takes to get to `on_read()`, you only know that you'll get there.

Asynchronous client

Things are a bit more complicated now but are definitely manageable. And you'll have an application that doesn't block!



You should already be familiar with the following code:

```
#define MEM_FN(x)          boost::bind(&self_type::x, shared_from_this())
#define MEM_FN1(x,y)      boost::bind(&self_type::x, shared_from_
this(),y)
#define MEM_FN2(x,y,z)    boost::bind(&self_type::x, shared_from_
this(),y,z)
class talk_to_svr : public boost::enable_shared_from_this<talk_to_svr>
, boost::noncopyable {
    typedef talk_to_svr self_type;
    talk_to_svr(const std::string & username)
        : sock_(service), started_(true), username_(username), timer_
(service) {}
    void start(ip::tcp::endpoint ep) {
        sock_.async_connect(ep, MEM_FN1(on_connect,_1));
    }
public:
    typedef boost::system::error_code error_code;
    typedef boost::shared_ptr<talk_to_svr> ptr;

    static ptr start(ip::tcp::endpoint ep, const std::string &
username) {
        ptr new_(new talk_to_svr(username));
        new_->start(ep);
        return new_;
    }
    void stop() {
        if ( !started_) return;
        started_ = false;
        sock_.close();
    }
    bool started() { return started_; }
    ...
private:
    size_t read_complete(const boost::system::error_code & err, size_t
bytes) {
        if ( err) return 0;
        bool found = std::find(read_buffer_, read_buffer_ + bytes,
'\n')
            < read_buffer_ + bytes;
        return found ? 0 : 1;
    }
}
```

```
private:
    ip::tcp::socket sock_;
    enum { max_msg = 1024 };
    char read_buffer_[max_msg];
    char write_buffer_[max_msg];
    bool started_;
    std::string username_;
    deadline_timer timer_;
};
```

You'll see an extra `deadline_timer timer_` function for ping the server; again, as you'll see, we'll ping the server at random times.

Now, the logic of the class is given as follows:

```
void on_connect(const error_code & err) {
    if ( !err)      do_write("login " + username_ + "\n");
    else           stop();
}
void on_read(const error_code & err, size_t bytes) {
    if ( err) stop();
    if ( !started() ) return;
    // process the msg
    std::string msg(read_buffer_, bytes);
    if ( msg.find("login ") == 0) on_login();
    else if ( msg.find("ping") == 0) on_ping(msg);
    else if ( msg.find("clients ") == 0) on_clients(msg);
}
void on_login() {
    do_ask_clients();
}
void on_ping(const std::string & msg) {
    std::istringstream in(msg);
    std::string answer;
    in >> answer >> answer;
    if ( answer == "client_list_changed") do_ask_clients();
    else postpone_ping();
}
void on_clients(const std::string & msg) {
    std::string clients = msg.substr(8);
    std::cout << username_ << ", new client list:" << clients ;
    postpone_ping();
}
```

In `on_read()`, nice touches are given in the first two lines of code. On the first line, if there's an error, we stop. On the second line, if we're stopped (as we were stopped before or just now), we return. Otherwise, if all's well, we process the incoming message.

And finally, `do_*` functions the following:

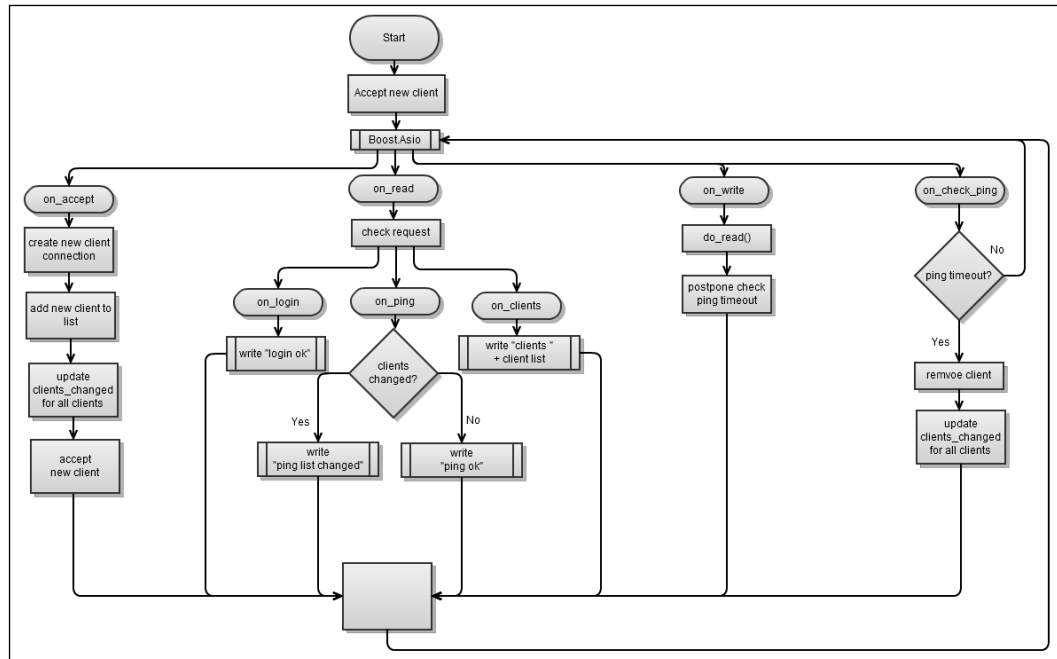
```
void do_ping() { do_write("ping\n"); }
void postpone_ping() {
    timer_.expires_from_now(boost::posix_time::millisec(rand() %
7000));
    timer_.async_wait( MEM_FN(do_ping));
}
void do_ask_clients() { do_write("ask_clients\n"); }
void on_write(const error_code & err, size_t bytes) { do_read(); }
void do_read() {
    async_read(sock_, buffer(read_buffer_),
MEM_FN2(read_complete, _1, _2), MEM_FN2(on_read, _1, _2));
}
void do_write(const std::string & msg) {
    if ( !started() ) return;
    std::copy(msg.begin(), msg.end(), write_buffer_);
    sock_.async_write_some( buffer(write_buffer_, msg.size()),
MEM_FN2(on_write, _1, _2));
}
```

Notice that every read operation will trigger a ping:

- When the read operation is complete, `on_read()` gets called
- `on_read()` dispatches to `on_login()`, `on_ping()`, or `on_clients()`
- Each of the functions either postpones a ping or asks for clients
- If we ask for clients when the read operation receives them, it postpones a ping

Asynchronous server

The diagram is pretty complex; from Boost.Asio, you'll see four arrows to `on_accept`, `on_read`, `on_write`, and `on_check_ping`. This basically means that you never know which asynchronous call finishes next, but you know for sure it's one of the four operations.



Now, we're asynchronous; we can stay single-threaded. Accepting clients is the easy part, as given in the following code snippet:

```

ip::tcp::acceptor acceptor(service, ip::tcp::endpoint(ip::tcp::v4(),
8001));
void handle_accept(talk_to_client::ptr client, const error_code & err)
{
    client->start();
    talk_to_client::ptr new_client = talk_to_client::new_();
    acceptor.async_accept(new_client->sock(),
        boost::bind(handle_accept,new_client,_1));
}
int main(int argc, char* argv[]) {
    talk_to_client::ptr client = talk_to_client::new_();
    acceptor.async_accept(client->sock(),
        boost::bind(handle_accept,client,_1));

    service.run();
}

```

The preceding code will asynchronously wait for new clients forever (each new client connection will then trigger another asynchronous wait).

We need to monitor for the `client list changed` event (a new client connects or one client gets disconnected), and notify all clients when this happens. Thus, we need to keep an array of clients, otherwise, there would be no need for this array unless you want to know all connected clients at a given time:

```
class talk_to_client; typedef boost::shared_ptr<talk_to_client>
client_ptr;
typedef std::vector<client_ptr> array;
array clients;
```

The skeleton of the connection class is as follows:

```
class talk_to_client : public boost::enable_shared_from_this<talk_to_
client>
    , boost::noncopyable {
    talk_to_client() { ... }
public:
    typedef boost::system::error_code error_code;
    typedef boost::shared_ptr<talk_to_client> ptr;
    void start() {
        started_ = true;
        clients.push_back( shared_from_this());
        last_ping = boost::posix_time::microsec_clock::local_time();
        do_read(); // first, we wait for client to login
    }
    static ptr new_() { ptr new_(new talk_to_client); return new_; }
    void stop() {
        if ( !started_) return;
        started_ = false;
        sock_.close();
        ptr self = shared_from_this();
        array::iterator it = std::find(clients.begin(), clients.end(),
self);
        clients.erase(it);
        update_clients_changed();
    }
    bool started() const { return started_; }
    ip::tcp::socket & sock() { return sock_;}
    std::string username() const { return username_; }
    void set_clients_changed() { clients_changed_ = true; }
    ...
}
```

```
private:
    ip::tcp::socket sock_;
    enum { max_msg = 1024 };
    char read_buffer_[max_msg];
    char write_buffer_[max_msg];
    bool started_;
    std::string username_;
    deadline_timer timer_;
    boost::posix_time::ptime last_ping;
    bool clients_changed_;
};
```

I'm calling the connection class either `talk_to_client` or `talk_to_server` to make it more clear what I'm talking about.

You should be used to the preceding code by now; it's similar to what we used for the client application. We do have an extra function, `stop()`, which removes a client connection from the client's array.

The server continuously waits for asynchronous read operations:

```
void on_read(const error_code & err, size_t bytes) {
    if ( err ) stop();
    if ( !started() ) return;
    std::string msg(read_buffer_, bytes);
    if ( msg.find("login ") == 0 ) on_login(msg);
    else if ( msg.find("ping") == 0 ) on_ping();
    else if ( msg.find("ask_clients") == 0 ) on_clients();
}
void on_login(const std::string & msg) {
    std::istringstream in(msg);
    in >> username_ >> username_;
    do_write("login ok\n");
    update_clients_changed();
}
void on_ping() {
    do_write(clients_changed_ ? "ping client_list_changed\n" : "ping
ok\n");
    clients_changed_ = false;
}
```

```

void on_clients() {
    std::string msg;
    for(array::const_iterator b =clients.begin(),e =clients.end(); b
!= e; ++b)
        msg += (*b)->username() + " ";
    do_write("clients " + msg + "\n");
}

```

The code is straight forward; one thing to notice is that when a new client logs in, we call `update_clients_changed()`, which sets `clients_changed_` to true for all clients.

Once it gets a request, it answers it right way, as given in the following code snippet:

```

void do_ping() { do_write("ping\n"); }
void do_ask_clients() { do_write("ask_clients\n"); }
void on_write(const error_code & err, size_t bytes) { do_read(); }
void do_read() {
    async_read(sock_, buffer(read_buffer_),
               MEM_FN2(read_complete,_1,_2), MEM_FN2(on_read,_1,_2));
    post_check_ping();
}
void do_write(const std::string & msg) {
    if ( !started() ) return;
    std::copy(msg.begin(), msg.end(), write_buffer_);
    sock_.async_write_some( buffer(write_buffer_, msg.size()),
                           MEM_FN2(on_write,_1,_2));
}
size_t read_complete(const boost::system::error_code & err, size_t
bytes) {
    // ... as before
}

```

At the end of each write operation, `on_write()` is called, which triggers another asynchronous read, and so the wait-for-request-and-answer-it cycle continues until the client disconnects or times out.

As each read begins, we asynchronously wait 5 seconds to see if the client times out. If so, we close its connection:

```
void on_check_ping() {
    ptime now = microsec_clock::local_time();
    if ( (now - last_ping).total_milliseconds() > 5000)
        stop();
    last_ping = boost::posix_time::microsec_clock::local_time();
}
void post_check_ping() {
    timer_.expires_from_now(boost::posix_time::millisec(5000));
    timer_.async_wait( MEM_FN(on_check_ping));
}
```

That's the whole server. You can run it, and put it to work!

In the code, I have shown you what we have seen in this chapter, as I have trimmed the code a bit to make it easier to grasp; for instance, I haven't shown most console messages, even though they exist in the code accompanying the book. I suggest you run the examples yourself, as seeing the code from top to bottom will solidify your understanding of the applications shown in this chapter.

Summary

We've seen how to write a few basic client/server applications. We've avoided pitfalls such as memory leaks and deadlocks. All the programs are meant to be skeletons that you can extend and adapt for your needs.

In the following chapter, we'll get a deeper understanding of the synchronous versus asynchronous differences when using Boost.Asio, and see how you can plug your own asynchronous operations as well.

5

Synchronous Versus Asynchronous

The author of Boost.Asio has done a wonderful job at giving you the option to choose what suits your application best by going synchronous or asynchronous.

In the previous chapter, we've seen skeletons of each type of application, such as synchronous client, synchronous server, asynchronous client, and asynchronous server. You can use each as a base for your applications. In case you need to delve into more detail about each type of application, read ahead.

Mixing synchronous and asynchronous programming

The Boost.Asio library allows you to mix synchronous and asynchronous programming. Personally, I think it's a bad idea, but Boost.Asio, just like C++, allows you to shoot yourself in the foot, if this is what you want.

You could easily fall into this trap, usually, when you have an asynchronous application. For instance, in response to an asynchronous write operation, lets say, you do a synchronous read operation:

```
io_service service;
ip::tcp::socket sock(service);
ip::tcp::endpoint ep( ip::address::from_string("127.0.0.1"), 8001);
void on_write(boost::system::error_code err, size_t bytes) {
    char read_buff[512];
    read(sock, buffer(read_buff));
}
async_write(sock, buffer("echo"), on_write);
```

For sure, the synchronous `read` operation will block the current thread, thus, any other pending asynchronous operations are put on hold (for this thread). This is a bad code and can lead to the application being unresponsive or being blocked altogether (the whole point of going asynchronous is to avoid blocking, thus, by doing a synchronous operations you negate that).

When you have a synchronous application, it's pretty unlikely that you'll do asynchronous `read` or `write` operations, since thinking synchronously already means thinking in a linear manner (do A, then B, then C, and so on).

The only scenario I can think of where synchronous and asynchronous might work together well is when the synchronous operations are completely separated from the asynchronous operations, for instance, synchronous networking and asynchronous input from or output to a database.

Passing client to server messages and vice versa

A very important part of a successful client/server application is passing messages back and forth (server to client and client to server). You need to specify what identifies a message. In other words, when reading an incoming message, how do I know it's been fully read?

It's up to you to identify the end of the message (the start of the message is easy, as it's the first byte passed at the end of the previous message), but do make sure it's easy and consistent.

You can:

- Have fixed-size messages (it's not such a good idea; what will happen when you need to send more data?)
- Have a specific character that ends the message, such as `'\n'` or `'\0'`
- Specify the message length as the prefix of the message and so on

Throughout the book, I have chosen to go for "end each message in `\n`". So, reading a message will always be like the following code snippet:

```
char buff_[512];
// synchronous read
read(sock_, buffer(buff_),
      boost::bind(&read_complete, this, _1, _2));
// asynchronous read
async_read(sock_, buffer(buff_),
```

```

MEM_FN2(read_complete, _1, _2), MEM_FN2(on_read, _1, _2));
size_t read_complete(const boost::system::error_code & err, size_t
bytes) {
    if ( err) return 0;
    already_read_ = bytes;
    bool found = std::find(buff_, buff_ + bytes, '\n') < buff_ +
bytes;
    // we read one-by-one until we get to enter, no buffering
    return found ? 0 : 1;
}

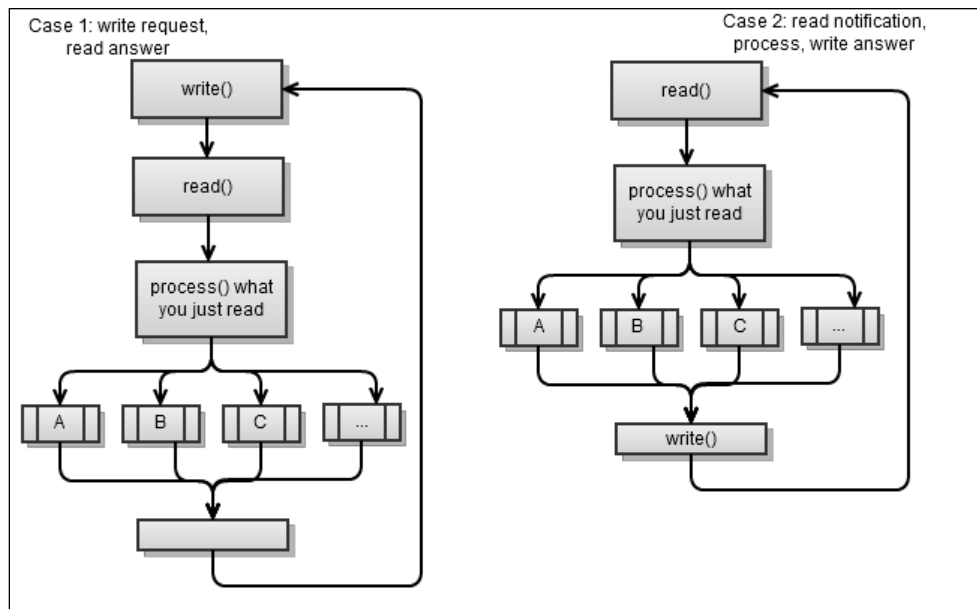
```

I leave specifying the message length as the prefix of the message as an exercise for the reader; it's pretty easy.

Synchronous I/O in client applications

A synchronous client will usually fall into one of the two cases:

- It requests something from the server, reads answer, and processes it. Then, ask something else, and so on. This is, in fact, what the synchronous client in the previous chapter is like.
- Read incoming message from the server, process it, then write answer. Then, read next incoming message, and so on.



Both scenarios use a make-request-read-answer strategy. In other words, one party makes a request to which the other party will answer back. This is an easy way to implement a client/server application, and that's what I highly recommend you do.

You can always create a Mambo Jambo client/server, where each party writes whenever it pleases, but it will very likely turn into a recipe for disaster (how will you know what happened when the client or server blocks?).

The preceding scenarios might look alike, but they are very different:

- In the former case, the server reacts to requests (the server waits for requests from clients and answers them). This is a pull-like connection, where the client pulls what it needs from the server.
- In the latter case, the server sends events to the client to which the client reacts. This is a push-like connection, where the server pushes notifications/events to the clients.

You'll mostly work on pull-like client/server applications, which are easier to develop, and also, are usually the norm.

You can also mix up pull requests (client to server) with push requests (server to client), however, this is very complicated, and you would better avoid it. The problem with mixing the two is, if you use the make-request-read-answer strategy, the following can happen:

- Client writes (makes request)
- Server writes (sends notification to client)
- Client reads what server wrote, and interprets it as an answer to its request
- Server blocks waiting for an answer from the client, which will come whenever the client makes a new request
- Client writes (makes a new request)
- Server will interpret that request as the answer it was waiting for
- Client will block (server won't send any reply back, because it interpreted the client request as the answer to its notification)

In a pull-like client/server application, it's easy to avoid the preceding scenario. You can simulate push-like behaviour by implementing a ping process, where the client pings the server, lets say, every 5 seconds. The server can answer with something such as `ping ok` if there's nothing to report or `ping [event_name]` if there's an event to report. Then the client can initiate a new request to handle that event.

To reiterate, the former scenario is the synchronous client application from the previous chapter. Its main loop is:

```
void loop() {
    // read answer to our login
    write("login " + username_ + "\n");
    read_answer();
    while ( started_ ) {
        write_request();
        read_answer();
        ...
    }
}
```

Lets change this to match the latter scenario:

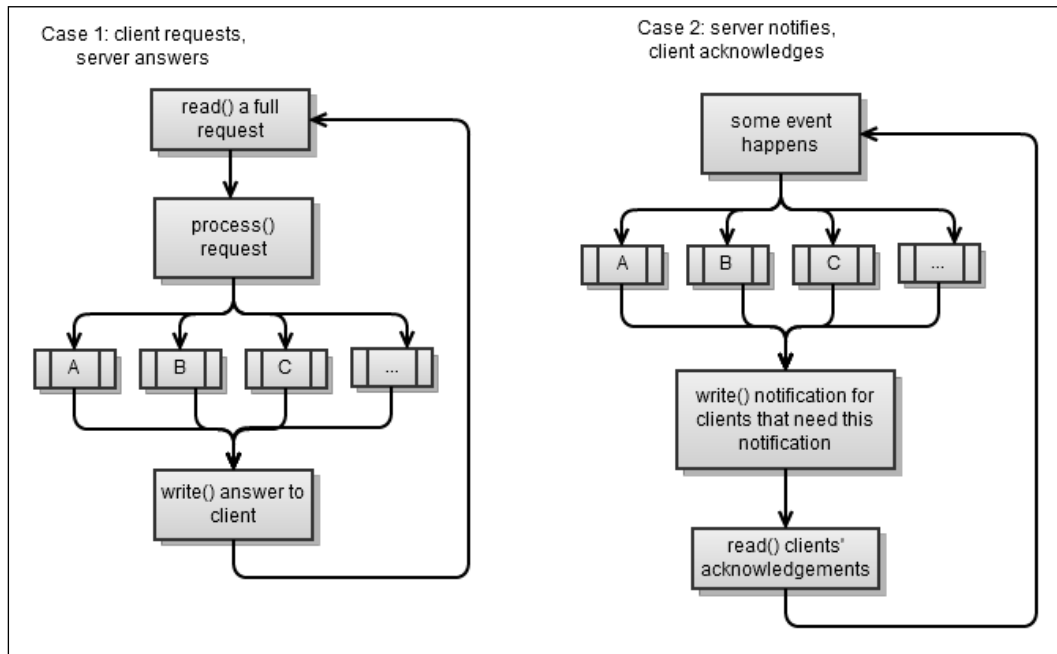
```
void loop() {
    while ( started_ ) {
        read_notification();
        write_answer();
    }
}

void read_notification() {
    already_read_ = 0;
    read(sock_, buffer(buff_),
        boost::bind(&talk_to_svr::read_complete, this, _1, _2));
    process_notification();
}

void process_notification() {
    // ... see what the notification is, and prepare answer
}
```

Synchronous I/O in server applications

Servers, like clients, are divided into two cases; they match scenarios one and two from the previous section. Again, both scenarios use a make-request-read-answer strategy.



The first scenario is the synchronous server we've implemented in the previous chapter. Reading a full request is not easy when you're synchronous, since you want to avoid blocking (you always read as much as you can):

```
void read_request() {
    if ( sock_.available())
        already_read_ += sock_.read_some(
            buffer(buff_ + already_read_, max_msg - already_read_));
}
```

Once a message has been fully read, just process it and answer the client:

```
void process_request() {
    bool found_enter = std::find(buff_, buff_ + already_read_, '\n')
        < buff_ + already_read_;
    if ( !found_enter)
        return; // message is not full
    size_t pos = std::find(buff_, buff_ + already_read_, '\n') -
buff_;
    std::string msg(buff_, pos);
    ...
    if ( msg.find("login ") == 0) on_login(msg);
    else if ( msg.find("ping") == 0) on_ping();
    else ...
}
```

If we wanted to have our server be a push server, we'd modify it as follows:

```
typedef std::vector<client_ptr> array;
array clients;
array notify;
std::string notify_msg;
void on_new_client() {
    // on a new client, we notify all clients of this event
    notify = clients;
    std::ostringstream msg;
    msg << "client count " << clients.size();
    notify_msg = msg.str();
    notify_clients();
}
void notify_clients() {
    for ( array::const_iterator b = notify.begin(), e = notify.end();
        b != e; ++b) {
        (*b)->sock_.write_some(notify_msg);
    }
}
```

The `on_new_client()` function is one of the events, where we need to notify all clients. `notify_clients` is the function that will notify all clients interested in an event. It sends the message but does not wait for a reply from each client, since that would be blocking. When a reply is coming from a client, the client can tell us what the reply is for (then we can process it correctly).

Threading in a synchronous server

This is a very important consideration: how many threads do we allocate to handle clients?

For a synchronous server, we'll need at least one thread that will handle new connections:

```
void accept_thread() {
    ip::tcp::acceptor acceptor(service, ip::tcp::endpoint(ip::tcp::
: v4(), 8001));
    while ( true) {
        client_ptr new_( new talk_to_client);
        acceptor.accept(new_ -> sock());
        boost::recursive_mutex::scoped_lock lk(cs);
        clients.push_back(new_);
    }
}
```

For the existing clients:

- We can go single-threaded. This is the easiest, and that's what I chose when I implemented the synchronous server in *Chapter 4, Synchronous Server*. It will easily handle 100 to 200 concurrent clients and sometimes maybe more, which is enough for the vast majority of time.
- We can have a thread per client. It's very rarely a good option; it will waste a lot of threads making debugging sometimes difficult, and while it will probably handle more than 200 concurrent users, it will reach its limit rather soon afterwards.
- We can have a fixed number of threads to handle existing clients.

The third option is pretty hard to implement in a synchronous server; the whole `talk_to_client` class has become thread-safe. Then, you'll need a mechanism to know which thread handles which clients. For this, you have two options:

- Assign specific clients to a specific thread; for instance, thread 1 handles the first 20 clients, thread 2 handles clients 21 to 40, and so on. When a client is in use (we're waiting for something that is blocking from that client), take it out of the array of existing clients. After we've dealt with it, put it back in the list. Each thread will loop through the existing clients, and take the first client with a full request (we've read a complete incoming message from the client), and answer it.

- The server can become unresponsive:
 - In the former case, several clients handled by the same thread made requests at the same time, since one thread deals with one request at a time. There's nothing we can do in this scenario.
 - In the latter case, when we get a number of concurrent requests higher than the number of threads. In this case, we can simply create new threads to handle the load.

The following code snippet, which is similar to the original `answer_to_client` function, shows how the latter scenario can be implemented:

```
struct talk_to_client : boost::enable_shared_from_this<talk_to_client>
{
    ...
    void answer_to_client() {
        try {
            read_request();
            process_request();
        } catch ( boost::system::system_error& ) {
            stop();
        }
    }
};
```

We'll modify it like the following code snippet:

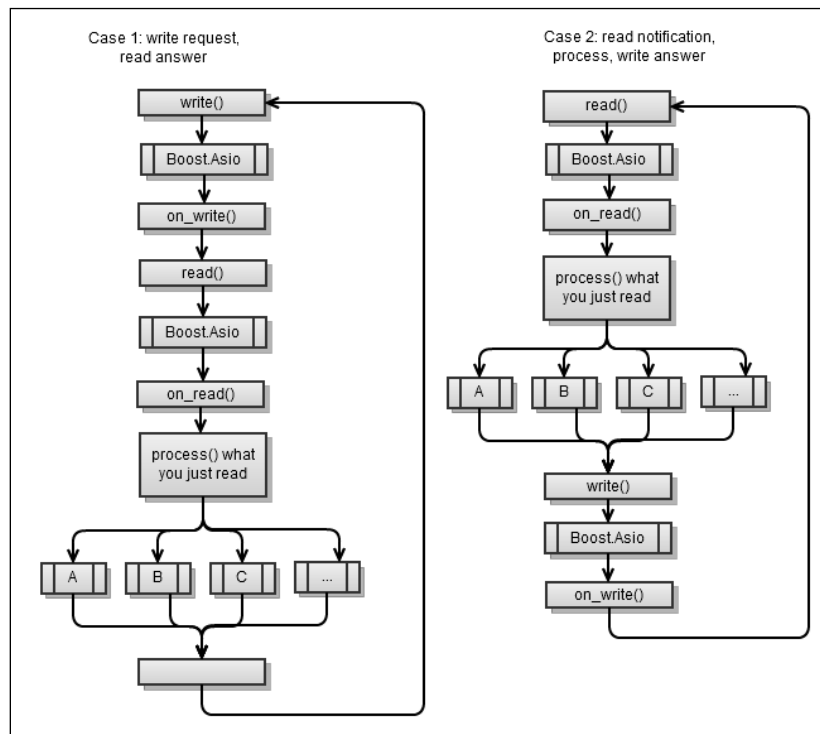
```
struct talk_to_client : boost::enable_shared_from_this<talk_to_client>
{
    boost::recursive_mutex cs;
    boost::recursive_mutex cs_ask;
    bool in_process;
    void answer_to_client() {
        { boost::recursive_mutex::scoped_lock lk(cs_ask);
          if ( in_process )
              return;
          in_process = true;
        }
        { boost::recursive_mutex::scoped_lock lk(cs);
          try {
              read_request();
              process_request();
          }
        }
    }
};
```

```
    } catch ( boost::system::system_error& ) {
        stop();
    }
}
{ boost::recursive_mutex::scoped_lock lk(cs_ask);
  in_process = false;
}
};
```

As long as we're processing a client, its `in_process` instance is set to `true`, and other threads will ignore that client. The added bonus is that the `handle_clients_thread()` function does not need to be modified; you can simply create as many `handle_clients_thread()` functions as you wish.

Asynchronous I/O in client applications

The main workflow is somewhat similar to that of the synchronous client applications, with the difference that Boost.Asio sits in between every `async_read` and `async_write` requests.



The first scenario is what I've implemented as asynchronous client in *Chapter 4, Client and Server*. Remember that at the end of each asynchronous operation, I start another asynchronous operation, so that the `service.run()` function doesn't end.

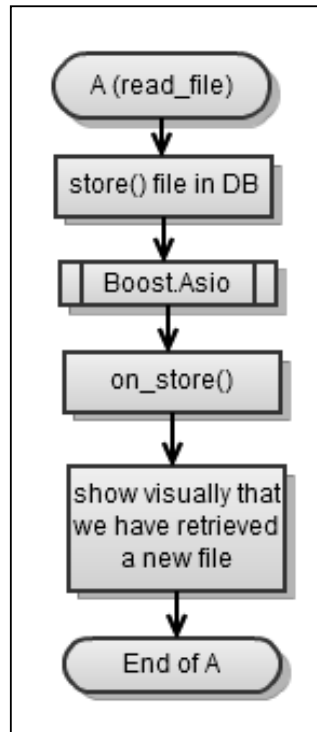
To change this to the second scenario, you'll use the following code snippet:

```
void on_connect() {
    do_read();
}
void do_read() {
    async_read(sock_, buffer(read_buffer_),
               MEM_FN2(read_complete, _1, _2), MEM_FN2(on_read, _1, _2));
}
void on_read(const error_code & err, size_t bytes) {
    if (err) stop();
    if (!started()) return;
    std::string msg(read_buffer_, bytes);
    if (msg.find("clients") == 0) on_clients(msg);
    else ...
}
void on_clients(const std::string & msg) {
    std::string clients = msg.substr(8);
    std::cout << username_ << ", new client list:" << clients ;
    do_write("clients ok\n");
}
```

Note that as soon as you are successfully connected, you start reading from the server. Each `on_[event]` function will end with us writing an answer to the server.

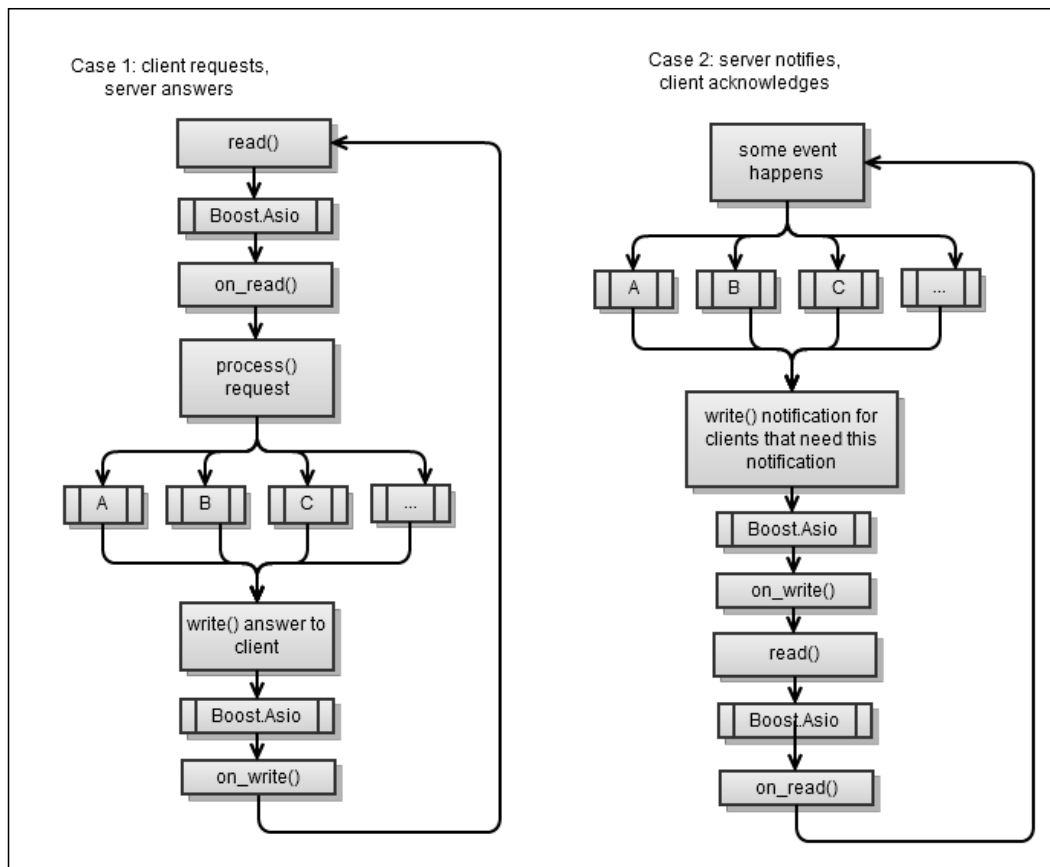
The beauty of going asynchronous is that you can mix the I/O networking operations with any other asynchronous operations, using Boost.Asio to orchestrate it all. Even though the flow is not as clear as the synchronous flow, you can still virtually think of it in a synchronous manner.

Say, you're reading files from a web server and storing them into a database (asynchronously). You can virtually think about it, as shown in the following flowchart:



Asynchronous I/O in server applications

Here are the ubiquitous two cases, scenario one (pull) and scenario 2 (push):



The first scenario is the asynchronous server I implemented in *Chapter 4, Client and Server* again. At the end of each asynchronous operation, I start another asynchronous operation, so that `service.run()` doesn't end.

Here's the skeleton code, which is trimmed down. The following are all members in `talk_to_client` class:

```
void start() {
    ...
    do_read(); // first, we wait for client to login
}
void on_read(const error_code & err, size_t bytes) {
    std::string msg(read_buffer_, bytes);
    if ( msg.find("login ") == 0) on_login(msg);
    else if ( msg.find("ping") == 0) on_ping();
    else ...
}
void on_login(const std::string & msg) {
    std::istringstream in(msg);
    in >> username_ >> username_;
    do_write("login ok\n");
}
void do_write(const std::string & msg) {
    std::copy(msg.begin(), msg.end(), write_buffer_);
    sock_.async_write_some( buffer(write_buffer_, msg.size()),
                           MEM_FN2(on_write, _1, _2));
}
void on_write(const error_code & err, size_t bytes) {
    do_read();
}
```

In a nutshell, we're always waiting for a read operation, and as soon as that happens, we process it and answer back to the client.

Lets modify the preceding code into a push server:

```
void start() {
    ...
    on_new_client_event();
}
void on_new_client_event() {
    std::ostringstream msg;
    msg << "client count " << clients.size();
    for ( array::const_iterator b = clients.begin(), e = clients.
end();
        b != e; ++b)
        (*b)->do_write(msg.str());
}
```

```

void on_read(const error_code & err, size_t bytes) {
    std::string msg(read_buffer_, bytes);
    // basically here, we only acknowledge
    // that our clients received our notifications
}
void do_write(const std::string & msg) {
    std::copy(msg.begin(), msg.end(), write_buffer_);
    sock_.async_write_some( buffer(write_buffer_, msg.size()),
                           MEM_FN2(on_write, _1, _2));
}
void on_write(const error_code & err, size_t bytes) {
    do_read();
}

```

As soon as an event happens, lets say, `on_new_client_event`, all the clients that need to be informed of that event are being sent a message. When they reply back, we simply see that they acknowledged receiving the event. Note that we'll never run out of asynchronous events to wait for (thus, `service.run()` doesn't end), since we're always waiting for new clients:

```

ip::tcp::acceptor acc(service, ip::tcp::endpoint(ip::tcp::v4(),
8001));
void handle_accept(talk_to_client::ptr client, const error_code & err)
{
    client->start();
    talk_to_client::ptr new_client = talk_to_client::new_();
    acc.async_accept(new_client->sock(), bind(handle_accept, new_
client, _1));
}

```

Threading in an asynchronous server

The asynchronous server I've shown you in *Chapter 4, Client and Server* is single-threaded, as everything happens in `main()`:

```

int main() {
    talk_to_client::ptr client = talk_to_client::new_();
    acc.async_accept(client->sock(), boost::bind(handle_
accept, client, _1));
    service.run();
}

```


The beauty of going asynchronous is the simplicity to move from single-threaded to multi-threaded. You can always go single-threaded until your concurrent clients goes over 200 or so. Then, to go from a single thread to 100 threads, you'll use the following code snippet:

```
boost::thread_group threads;
void listen_thread() {
    service.run();
}
void start_listen(int thread_count) {
    for ( int i = 0; i < thread_count; ++i)
        threads.create_thread( listen_thread );
}
int main(int argc, char* argv[]) {
    talk_to_client::ptr client = talk_to_client::new_();
    acc.async_accept(client->sock(), boost::bind(handle_
accept,client,_1));
    start_listen(100);
    threads.join_all();
}
```

Of course, once you go multi-threaded, you will have to think about thread safety. Even though you call `async_*` in thread A, its completion routine can be called in thread B (as long as thread B has called `service.run()`). That is not a problem in itself. As long as you follow the logical flow, that is, from `async_read()` to `on_read()`, from `on_read()` to `process_request`, from `process_request` to `async_write()`, from `async_write()` to `on_write()`, from `on_write()` to `async_read()`, and there are no public functions that are called on your `talk_to_client` class, even though different functions can be called on different threads, they will still be called in sequential order. Thus, no need for mutexes.

This, however, means that for a client, there can be only one asynchronous operation pending. If at some point, for a client, we have two pending asynchronous functions, you'll need mutexes. This is because the two pending operations might finish roughly at the same time, and we could end up having their completion handlers called simultaneously on two different threads. Therefore, there is a need for thread safety, and thus, mutexes.

In our asynchronous server, we actually have two pending operations at the same time:

```
void do_read() {
    async_read(sock_, buffer(read_buffer_),
               MEM_FN2(read_complete, _1, _2), MEM_FN2(on_read, _1, _2));
    post_check_ping();
}
void post_check_ping() {
    timer_.expires_from_now(boost::posix_time::millisec(5000));
    timer_.async_wait( MEM_FN(on_check_ping));
}
```

When doing a read operation, we'll asynchronously wait for the read operation to complete and for a deadline. Thus, there is a need for thread safety.

My advice is, if you plan to go multi-threaded, make your class thread-safe from the beginning. It usually won't hurt performance (you can profile-check it, of course). Also, if you plan to go multi-threaded, go that way from the beginning. This way, you'll discover possible problems early on. Once you discover a problem, first thing you need to check is: does it happen when there's a single thread running? If so, it's easy; just debug it. Otherwise, you probably forgot to lock (mutex) some function.

Since our example needs thread safety, I've modified `talk_to_client` to use mutexes. Also, we have an array of clients, which we reference a few times in the code that needs its own mutex as well.

Avoiding deadlock and memory corruption is not easy. Here's how I had to modify the `update_clients_changed()` function:

```
void update_clients_changed() {
    array copy;
    { boost::recursive_mutex::scoped_lock lk(clients_cs);
      copy = clients; }
    for( array::iterator b = copy.begin(), e = copy.end(); b != e;
        ++b)
        (*b)->set_clients_changed();
}
```

What you want to avoid is having two mutexes locked at the same time (which can lead to deadlocking). In our case, we don't want `clients_cs` and a client's `cs_mutex` to be locked at the same time.

You can check out the full application in the code that comes with the book.

Asynchronous operations

Boost.Asio also allows you to run any of your functions asynchronously. Just use the following code snippet:

```
void my_func() {
    ...
}
service.post(my_func);
```

That will make sure that `my_func` is called in one of the threads that called `service.run()`. You can also run a function asynchronously and have a completion handler, which will notify you when the function completes. The pseudocode would look like the following code snippet:

```
void on_complete() {
    ...
}
void my_func() {
    ...
    service.post(on_complete);
}
async_call(my_func);
```

There is no `async_call` function, however, you'll have to create your own. Fortunately, it's not that hard. Refer to the following code snippet:

```
struct async_op : boost::enable_shared_from_this<async_op>, ... {
    typedef boost::function<void(boost::system::error_code)>
completion_func;
    typedef boost::function<boost::system::error_code ()> op_func;
    struct operation { ... };
    void start() {
        { boost::recursive_mutex::scoped_lock lk(cs_);
          if ( started_ ) return; started_ = true; }
        boost::thread t( boost::bind(&async_op::run,this));
    }
    void add(op_func op, completion_func completion, io_service
&service) {
        self_ = shared_from_this();
        boost::recursive_mutex::scoped_lock lk(cs_);
        ops_.push_back( operation(service, op, completion));
        if ( !started_ ) start();
    }
}
```

```

    void stop() {
        boost::recursive_mutex::scoped_lock lk(cs_);
        started_ = false; ops_.clear();
    }
private:
    boost::recursive_mutex cs_;
    std::vector<operation> ops_; bool started_; ptr self_;
};

```

The `async_op` function creates a background thread, which will run (`run()`) all the asynchronous operations you add (`add()`) to it. I kept things easy, as each operation consists of these things:

- A function to call asynchronously.
- A completion function to call when the first function has completed.
- The `io_service` instance that will execute the completion function. This is where you will be notified of the completion. Refer to the following code snippet:

```

struct async_op : boost::enable_shared_from_this<async_op>
    , private boost::noncopyable {
    struct operation {
        operation(io_service & service, op_func op, completion_
func completion)
            : service(&service), op(op), completion(completion)
            , work(new io_service::work(service))
        {}
        operation() : service(0) {}
        io_service * service;
        op_func op;
        completion_func completion;
        typedef boost::shared_ptr<io_service::work> work_ptr;
        work_ptr work;
    };
    ...
};

```

Internally, they are held in the operation structure. Notice that while an operation is pending, we construct a `io_service::work` instance in operation's constructor, so that `service.run()` doesn't end until we've completed our asynchronous call (while the `io_service::work` instance is alive, `service.run()` will consider that it has work to do). Refer to the following code snippet:

```
struct async_op : ... {
    typedef boost::shared_ptr<async_op> ptr;
    static ptr new_() { return ptr(new async_op); }
    ...
    void run() {
        while ( true) {
            { boost::recursive_mutex::scoped_lock lk(cs_);
              if ( !started_) break; }
            boost::this_thread::sleep( boost::posix_
time::millisec(10));
            operation cur;
            { boost::recursive_mutex::scoped_lock lk(cs_);
              if ( !ops_.empty()) {
                  cur = ops_[0]; ops_.erase( ops_.begin());
              }
            }
            if ( cur.service)
                cur.service->post( boost::bind( cur.completion, cur.op()
));
        }
        self_.reset();
    }
};
```

The `run()` function is the background thread; it simply sees if there's work to do, and if so, executes the asynchronous functions one by one. At the end of each call, it calls the corresponding completion function.

To test it, we'll create a function `compute_file_checksum` to be asynchronously executed:

```
size_t checksum = 0;
boost::system::error_code compute_file_checksum(std::string file_name)
{
```

```

HANDLE file = ::CreateFile(file_name.c_str(), GENERIC_READ, 0, 0,
    OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL | FILE_FLAG_OVERLAPPED, 0);
windows::random_access_handle h(service, file);
long buff[1024];
checksum = 0;
size_t bytes = 0, at = 0;
boost::system::error_code ec;
while ( (bytes = read_at(h, at, buffer(buff), ec)) > 0) {
    at += bytes; bytes /= sizeof(long);
    for ( size_t i = 0; i < bytes; ++i)
        checksum += buff[i];
}
return boost::system::error_code(0, boost::system::generic_
category());
}
void on_checksum(std::string file_name, boost::system::error_code) {
    std::cout << "checksum for " << file_name << "=" << checksum <<
std::endl;
}
int main(int argc, char* argv[]) {
    std::string fn = "readme.txt";
    async_op::new_()->add( service, boost::bind(compute_file_
checksum,fn),
                                boost::bind(on_checksum,fn,_1));

    service.run();
}

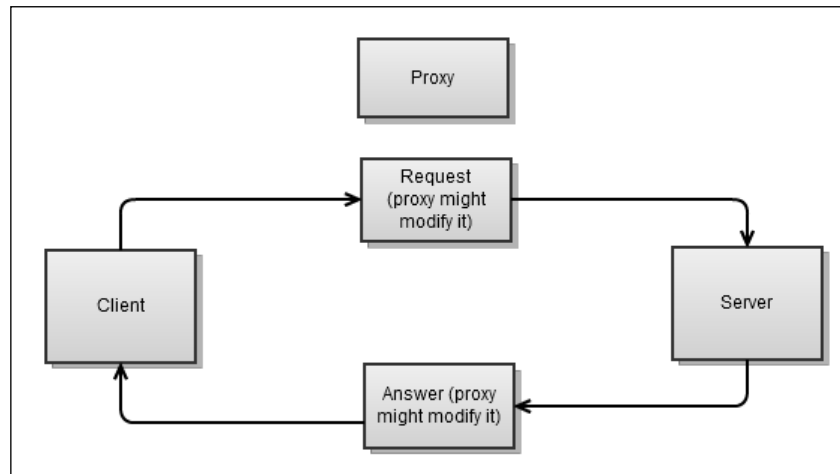
```

Note that what I've shown you is just a possible implementation of calling a function asynchronously. Instead of implementing the background thread, like I did, you can use an internal `io_service` instance to which you would post (`post()`) the asynchronous functions to call. This is an exercise for the reader.

You can also extend the class to allow showing progress of an asynchronous operation (for example, in percent). In such a case, you could show that progress in a progress bar, in the main thread.

Implementing proxies

A proxy usually sits between a client and a server. It takes a request from a client, might modify it, and forwards it to the server. It then takes the answer from the server, might modify it, and forwards it to the client.



What's special about a proxy, for our purposes, is that for each connection, you'll have two sockets, one to the client and the other to the server. This complicates implementing a proxy quite a bit.

Implementing the proxy as a synchronous application will be more complicated than to have it asynchronous; data could be coming from both ends (client and server) at the same time, and data might be going to both ends (client and server) at the same time. That means if we were to go synchronous, we could end up being blocked in a `read()` from or `write()` to a party while we need to `read()` from or `write()` to the other party, which means we'll be unresponsive on one end.

Consider the following items as a simple example of an asynchronous proxy:

- In our scenario, we know both connections at construction. This is not always the case, for a web proxy, for instance, the client tells us the address of the server.
- For the sake of simplicity, it's not thread-safe. Refer to the following code snippet:

```
class proxy : public boost::enable_shared_from_this<proxy> {
    proxy(ip::tcp::endpoint ep_client, ip::tcp::endpoint ep_
server) : ... {}
public:
    static ptr start(ip::tcp::endpoint ep_client,
ip::tcp::endpoint ep_svr) {
        ptr new_(new proxy(ep_client, ep_svr));
        // ... connect to both endpoints
        return new_;
    }
    void stop() {
        // ... stop both connections
    }
    bool started() { return started_ == 2; }
private:
    void on_connect(const error_code & err) {
        if ( !err) {
            if ( ++started_ == 2) on_start();
        } else stop();
    }
    void on_start() {
        do_read(client_, buff_client_);
        do_read(server_, buff_server_);
    }
    ...
private:
    ip::tcp::socket client_, server_;
    enum { max_msg = 1024 };
    char buff_client_[max_msg], buff_server_[max_msg];
    int started_;
};
```


It's a very simple proxy. When connected on both ends, it starts reading on both connections (function `on_start()`):

```
class proxy : public boost::enable_shared_from_this<proxy> {
...
    void on_read(ip::tcp::socket & sock, const error_code& err, size_t
bytes) {
        char * buff = &sock == &client_ ? buff_client_ : buff_server_;
        do_write(&sock == &client_ ? server_ : client_, buff, bytes);
    }
    void on_write(ip::tcp::socket & sock, const error_code &err,
size_t bytes){
        if ( &sock == &client_) do_read(server_, buff_server_);
        else do_read(client_, buff_client_);
    }
    void do_read(ip::tcp::socket & sock, char* buff) {
        async_read(sock, buffer(buff, max_msg),
MEM_FN3(read_complete,ref(sock),_1,_2),
MEM_FN3(on_read,ref(sock),_1,_2));
    }
    void do_write(ip::tcp::socket & sock, char * buff, size_t size) {
        sock.async_write_some(buffer(buff,size),
MEM_FN3(on_write,ref(sock),_1,_2));
    }
    size_t read_complete(ip::tcp::socket & sock,
const error_code & err, size_t bytes) {
        if ( sock.available() > 0) return sock.available();
        return bytes > 0 ? 0 : 1;
    }
};
```

On each successful read (`on_read`), it forwards the message to the other party. Once the message has been forwarded successfully (`on_write`), we read it again from the original party.

To put this to work, use the following code snippet:

```
int main(int argc, char* argv[]) {
    ip::tcp::endpoint ep_c( ip::address::from_string("127.0.0.1"),
8001);
    ip::tcp::endpoint ep_s( ip::address::from_string("127.0.0.1"),
8002);
    proxy::start(ep_c, ep_s);
    service.run();
}
```

You will notice that I'm reusing the buffers (`buff_client_` and `buff_server_`) for both reading and writing. This re-usage is okay, because a `read` message from a client is written to server before a new message is read from a client and vice versa. This also means that this particular implementation suffers from a responsiveness problem. While we're in the process of writing to party B, we're not reading from party A (we will restart reading from party A once the writing to party B is complete). You can modify the implementation to overcome this by doing the following:

- You should use multiple read buffers
- On each successful `read` operation, besides asynchronously writing to the other party, do an extra asynchronous `read` (into a new buffer)
- On each successful `write` operation, destroy (or reuse) the buffer

I will leave this exercise to you.

Summary

There are many things to consider when choosing to go synchronous or asynchronous. First off, avoid mixing them.

In this chapter, we've seen:

- How easy it is to implement, test, and debug each type of application
- How threading affects your application
- How the application behavior (pull-like or push-like) affects its implementation
- How you can plug in your own asynchronous operations when you go asynchronous

Following, we're about to see a few not-so-well-known features of Boost.Asio, and my Boost.Asio favorite feature, co-routines, which allows you to grab the pros of going asynchronous with close to none of its cons!

6

Boost.Asio – Other Features

Here we'll see some of the not-so-well-known features of Boost.Asio. The `std` streams and `streambuf` objects are sometimes a bit more complicated to use, but as you'll see, they bring their own benefits to the table. Finally, you'll see a rather late entry to Boost.Asio's co-routines, which allow you to have a code that is asynchronous but is easy to read (as if it was synchronous). It's quite an amazing feature!

std streams and std buffer I/O

You should be familiar with STL streams and STL `streambuf` objects in order to read this section.

Boost.Asio allows for two types of buffers when dealing with I/O:

- `boost::asio::buffer()`: This buffer surrounds a Boost.Asio operation (the buffers we use are passed to the Boost.Asio operation)
- `boost::asio::streambuf`: This buffer is derived from `std::streambuf`, and allows you to use STL streams together with networking

Throughout the book, you've mostly seen examples of the former case:

```
size_t read_complete(boost::system::error_code, size_t bytes){ ... }
char buff[1024];
read(sock, buffer(buff), read_complete);
write(sock, buffer("echo\n"));
```

You'll usually be happy with this. If you need more flexibility, you can go with `streambuf`.

Here's the easiest and the worst thing you can do with a `streambuf` object:

```
streambuf buf;  
read(sock, buf);
```

This will read until the `streambuf` object is full, and since the `streambuf` object can reallocate itself to accommodate more room, it will basically read until the connection is closed.

You can use `read_until` to read up to a sequence of characters:

```
streambuf buf;  
read_until(sock, buf, "\\n");
```

This will read until `'\\n'` is being read, then append that to the buffer, and exit the read function.

To write something from a `streambuf` object, you'll do something similar to the following:

```
streambuf buf;  
std::ostream out(&buf);  
out << "echo" << std::endl;  
write(sock, buf);
```

It's pretty straightforward; you construct an STL stream passing your `streambuf` object at construction, write to it the message you want to send, and then use `write` to send the contents of the buffer.

Boost.Asio and the STL streams

Boost.Asio has done a great job at integrating STL streams and networking. Namely, if you're already using the STL extensively, you already must have a lot of classes with overloaded operators, `>>` and `<<`. Reading and writing them to sockets will like be a walk in the park.

Say you have the following code snippet:

```
struct person {  
    std::string first_name, last_name;  
    int age;  
};  
std::ostream& operator<<(std::ostream & out, const person & p) {  
    return out << p.first_name << " " << p.last_name << " " << p.age;  
}
```

```
std::istream& operator>>(std::istream & in, person & p) {
    return in >> p.first_name >> p.last_name >> p.age;
}
```

Sending a person across the network is as easy as the following code snippet:

```
streambuf buf;
std::ostream out(&buf);
person p;
// ... initialize p
out << p << std::endl;
write(sock, buf);
```

The other party can read it quite easily as well:

```
read_until(sock, buf, "\n");
std::istream in(&buf);
person p;
in >> p;
```

The really good part when using a `streambuf` object, and of course, its corresponding `std::ostream` for writing or `std::istream` for reading, is that you end up writing code that feels natural:

- When writing something to be sent across the network, it's very likely that you'll have more than one piece of data. Thus, you'll end up appending data into the buffer. If that data is not a string, you'll need to convert it to a string first. All this happens by default when using the `<<` operator.
- Similarly, on the other party, when reading a message, you'll need to parse it, that is, read one piece of data at a time, and if the data is not a string, you'll need to convert it from a string. All this happens by default when you use the `>>` operator to read something.

Finally, here's a well-known, pretty cool trick; to dump the contents of the `streambuf` object to the console, use the following code snippet:

```
streambuf buf;
...
std::cout << &buf << std::endl; // dumps all content to the console
```

Similarly, to convert its contents to a string, use the following code snippet:

```
std::string to_string(streambuf &buf) {
    std::ostringstream out;
    out << &buf;
    return out.str();
}
```

The streambuf class

As I've said, `streambuf` derives from `std::streambuf`. Like `std::streambuf` itself, it's not copy-constructable.

In addition, it has a few extra functions, such as:

- `streambuf ([max_size,] [allocator])`: This function constructs a `streambuf` object. You can optionally specify a maximum buffer size and an allocator, which is used to allocate/deallocate memory when needed.
- `prepare (n)`: This function returns a sub-buffer, used to accommodate a contiguous sequence of n number of characters. It can be used for reading or writing. The result of this function can be used with any free function from Boost.Asio dealing with `read/write`, not just those dealing with `streambuf` objects.
- `data ()`: This function returns the whole buffer as a contiguous sequence of characters and is used for writing. The result of this function can be used with any free function from Boost.Asio dealing with writing, not just those dealing with `streambuf` objects.
- `consume (n)`: In this function, data is removed from the input sequence (from `read` operation).
- `commit (n)`: In this function, data is removed from the output sequence (from `write` operation) and added to the input sequence (for `read` operation).
- `size ()`: This function returns the size in characters, of the whole `streambuf` object.
- `max_size ()`: This function returns how many characters it can hold, at most.

Except for the last two functions, the other functions are not that easy to understand. First of all, most of the time, you'll send the `streambuf` instance as an argument to the `read/write` free functions, as shown in the following code snippet:

```
read_until(sock, buf, "\n"); // reads into buf
write(sock, buf); // writes from buf
```

If you send the whole buffer to a free function, as shown in the preceding snippet, the function will make sure it will increment the buffer's seek input and output pointers. In other words, if there's data to be read, you'll be able to read it. For example:

```
read_until(sock, buf, '\n');
std::cout << &buf << std::endl;
```

The preceding code snippet will dump what you just wrote from the socket. The following code snippet will not dump anything:

```
read(sock, buf.prepare(16), transfer_exactly(16) );
std::cout << &buf << std::endl;
```

The bytes are read, but the seek input pointer hasn't moved. You have to move it yourself, as given in the following code snippet:

```
read(sock, buf.prepare(16), transfer_exactly(16) );
buf.commit(16);
std::cout << &buf << std::endl;
```

Similarly, if you want to write from the `streambuf` object, if you use the free function `write`, use the following code snippet:

```
streambuf buf;
std::ostream out(&buf);
out << "hi there" << std::endl;
write(sock, buf);
```

The following code will send `hi there` three times:

```
streambuf buf;
std::ostream out(&buf);
out << "hi there" << std::endl;
for ( int i = 0; i < 3; ++i)
    write(sock, buf.data());
```

This happens because the buffer never consumes, as data still remains there. If you want to consume it, use the following code snippet:

```
streambuf buf;
std::ostream out(&buf);
out << "hi there" << std::endl;
write(sock, buf.data());
buf.consume(9);
```

In conclusion, you should prefer dealing with the whole `streambuf` instance. Use the preceding functions when you want fine-tuning.

Even though you can use the same `streambuf` instance for reading and writing, I recommend you use two separate ones, one for reading and one for writing. It will keep things simpler, clearer, and you'll avoid a lot of possible bugs.

The free functions that deal with streambuf objects

The following list shows what Boost.Asio free functions deal with `streambuf` objects as well:

- `read(sock, buf [, completion_function])`: This function reads from the socket into the `streambuf` object. The completion function is optional. If present, it's called after each successful read operation, and tells Boost.Asio if the operation is complete (if not, it continues to read). Its signature is `size_t completion(const boost::system::error_code &err, size_t bytes_transferred) ;`. When the completion function returns 0, we consider the read operation complete; if it returns a non-zero value, it indicates that it has maximum number of bytes to be read on the next call to the stream's `read_some`.
- `read_at(radom_stream, offset, buf [, completion_function])`: This function reads from a random stream. Note that this does not apply to sockets (since they don't model the random stream concept, they are forward only).
- `read_until(sock, buf, char | string | regex | match_condition)`: This function reads until a given condition matches. Either a `char` datatype has been read, or a string has been read, or a regex expression has matched in the read string so far, or the `match_condition` function says we should exit the function. The signature of `match_condition` is `pair<iterator, bool> match(iterator begin, iterator end) ;`, where `iterator` means `buffers_iterator<streambuf::const_buffers_type>`. If a match is found, you should return a pair (passed-end-of-match is set to `true`). If a match is not found, you should return pair (begin is set to `false`).
- `write(sock, buf [, completion_function])`: This function writes all contents of the `streambuf` object. The completion function is optional and its behavior is similar to the completion function for `read()`: return 0 when the write operation is complete, or return non-zero to indicate the number of bytes to be written on the next call to the stream's `write_some` function.

- `write_at(random_stream, offset, buf [, completion_function])`: This function writes to a random stream. Again, this does not apply to sockets.
- `async_read(sock, buf [, completion_function], handler)`: This function asynchronously counterparts to `read()`. The handler's signature is `void handler(const boost::system::error_code, size_t bytes)`.
- `async_read_at(radom_stream, offset, buf [, completion_function], handler)`: This function asynchronously counterparts to `read_at()`.
- `async_read_until(sock, buf, char | string | regex | match_condition, handler)`: This function asynchronously counterparts to `read_until()`.
- `async_write(sock, buf [, completion_function], handler)`: This function asynchronously counterparts to `write()`.
- `async_write_at(random_stream, offset, buf [, completion_function], handler)`: This function asynchronously counterparts to `write_at()`.

Lets say you want to read up to a vowel:

```
streambuf buf;
bool is_vowel(char c) {
    return c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u';
}
size_t read_complete(boost::system::error_code, size_t bytes) {
    const char * begin = buffer_cast<const char*>( buf.data());
    if ( bytes == 0) return 1;
    while ( bytes > 0)
        if ( is_vowel(*begin++)) return 0;
        else --bytes;
    return 1;
}
...
read(sock, buf, read_complete);
```

The thing to note here is accessing the buffer inside `read_complete()`, namely `buffer_cast<>` and `buf.data()`.

If you want to use `regex`, the example becomes much simpler:

```
read_until(sock, buf, boost::regex("^[aeiou]+") );
```

Or lets modify the example and you put the `match_condition` function to work:

```
streambuf buf;
bool is_vowel(char c) {
    return c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u';
}
typedef buffers_iterator<streambuf::const_buffers_type> iterator;
std::pair<iterator,bool> match_vowel(iterator b, iterator e) {
    while ( b != e)
        if ( is_vowel(*b++)) return std::make_pair(b, true);
    return std::make_pair(e, false);
}
...
size_t bytes = read_until(sock, buf, match_vowel);
```

When using `read_until`, there's a twist; you need to consider the number of bytes you've read, because the underlying buffer might have read more bytes (unlike when using `read()`). For example:

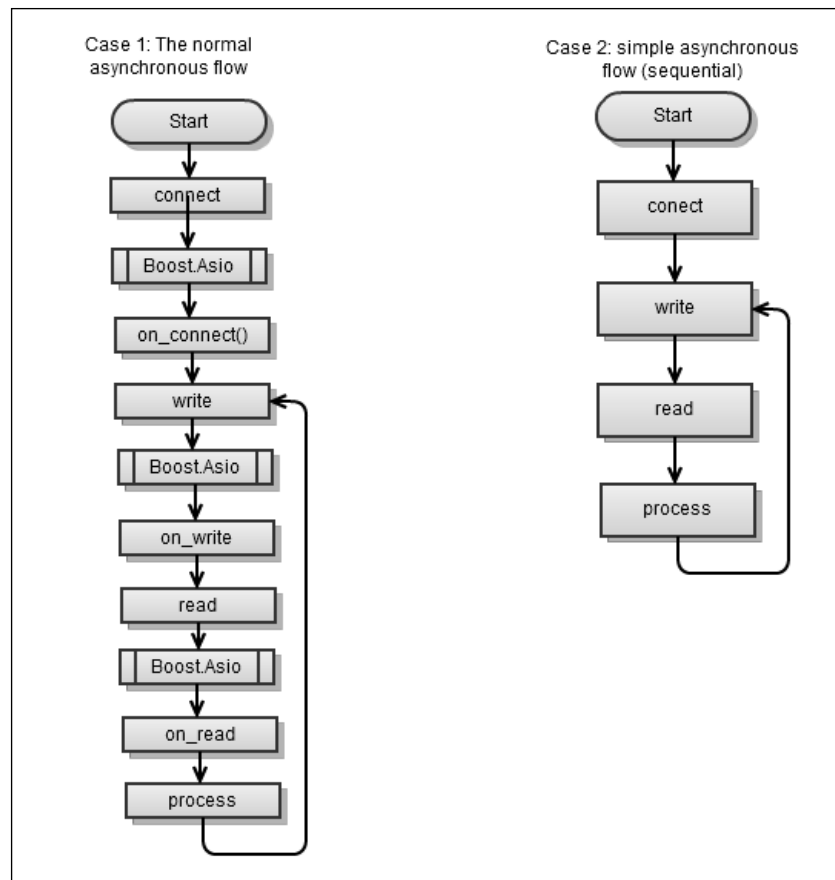
```
std::cout << &buf << std::endl;
```

The previous code snippet might dump more than the number of bytes read by `read_until`.

Co-routines

The author of Boost.Asio, around 2009-2010, implemented a very cool concept, co-routines, to help you design asynchronous applications even easier.

They allow you to have the best of both worlds, that is, write asynchronous applications and easily follow the flow of control, almost as if the application was written sequentially.



The normal flow is shown in case 1. Using co-routines, you'll get as close to case two as possible.

Simply put, a co-routine allows multiple entry points for suspending and resuming execution at certain locations within a function.

If you are to use co-routines, you'll need two header files that are only found in `boost/libs/asio/example/http/server4: yield.hpp` and `coroutine.hpp`. Here, Boost.Asio defines two pseudo-keywords (macros) and a class:

- `coroutine`: This class is to derive or use in your connection class in order to implement co-routines.
- `reenter(entry)`: This is the body of the co-routine. The argument `entry` is a pointer to a `coroutine` instance to be used as a block within the whole function.
- `yield code`: This executes a statement as part of the co-routine. Next time the function is entered, execution will start after this code.

To understand better, let's go for an example. We'll re-implement the application from *Chapter 4, Asynchronous Client*, which is a simple client that logs in, pings, and can tell you which other clients are logged.

The core code looks similar to the following code snippet:

```
class talk_to_svr : public boost::enable_shared_from_this<talk_to_svr>
                  , public coroutine, boost::noncopyable {
...
void step(const error_code & err = error_code(), size_t bytes = 0) {
    reenter(this) { for (;;) {
        yield async_write(sock_, write_buffer_, MEM_FN2(step, _1, _2) );
        yield async_read_until( sock_, read_buffer_, "\n", MEM_
FN2(step, _1, _2));
        yield service.post( MEM_FN(on_answer_from_server));
    }}
}
};
```

The first thing that's changed, instead of having lots of member functions, such as `connect()`, `on_connect()`, `on_read()`, `do_read()`, `on_write()`, `do_write()`, and so on, we have a single function called `step()`.

The body of the function is within `reenter(this) { for (;;) { }}`. You can think of `reenter(this)` as the code we executed last time, so we can execute the next code this time.

Within the `reenter` block, you'll see several `yield` statements. The first time you enter the function, the `async_write` function gets executed, second time the `async_read_until` function gets executed, third time the `service.post` function gets executed, and fourth time the `async_write` function gets executed, and so on.

You should never forget the `for(;;) {}` instance. Refer to the following code snippet:

```
void step(const error_code & err = error_code(), size_t bytes = 0) {
    reenter(this) {
        yield async_write(sock_, write_buffer_, MEM_FN2(step, _1, _2) );
        yield async_read_until( sock_, read_buffer_, "\n", MEM_
FN2(step, _1, _2));
        yield service.post( MEM_FN(on_answer_from_server));
    }
}
```

If we were to use the preceding code snippet the third time, we would have entered the function and would have executed `service.post`. Fourth time, when we enter the function, we go past `service.post` and execute nothing. Follow the same thing when executing for the fifth time and so on:

```
class talk_to_svr : public boost::enable_shared_from_this<talk_to_svr>
, public coroutine, boost::noncopyable {
    talk_to_svr(const std::string & username) : ... {}
    void start(ip::tcp::endpoint ep) {
        sock_.async_connect(ep, MEM_FN2(step, _1, 0) );
    }
    static ptr start(ip::tcp::endpoint ep, const std::string &
username) {
        ptr new_(new talk_to_svr(username)); new_->start(ep); return
new_;
    }
    void step(const error_code & err = error_code(), size_t bytes = 0)
{
    reenter(this) { for(;;) {
        if ( !started_ ) {
            started_ = true; std::ostream out(&write_buf_);
            out << "login " << username_ << "\n";
        }
        yield async_write(sock_, write_buf_, MEM_FN2(step, _1, _2)
);
        yield async_read_until( sock_, read_buf_, "\n", MEM_
FN2(step, _1, _2));
        yield service.post( MEM_FN(on_answer_from_server));
    }}
}
void on_answer_from_server() {
    std::istream in(&read_buf_); std::string word; in >> word;
```

```
        if ( word == "login") on_login();
        else if ( word == "ping") on_ping();
        else if ( word == "clients") on_clients();
        read_buf_.consume( read_buf_.size());
        if (write_buf_.size() > 0) service.post( MEM_FN2(step,error_
code(),0));
    }
    ...
private:
    ip::tcp::socket sock_; streambuf read_buf_, write_buf_;
    bool started_; std::string username_; deadline_timer timer_;
};
```

When we start the connection, `start()` gets called, which asynchronously connects to the server. When connection is completed, we enter `step()` for the first time. This is when we'll send our login message.

After that, we use `async_write`, then `async_read_until`, and process the message (`on_answer_from_server`).

In `on_answer_from_server`, we process the incoming message; we read the first word, and dispatch to the proper function. Then, ignore the remainder of the message (in case there was any):

```
class talk_to_svr : ... {
    ...
    void on_login() { do_ask_clients(); }
    void on_ping() {
        std::istream in(&read_buf_);
        std::string answer; in >> answer;
        if ( answer == "client_list_changed") do_ask_clients();
        else postpone_ping();
    }
    void on_clients() {
        std::ostringstream clients; clients << &read_buf_;
        std::cout << username_ << ", new client list:" << clients.
str();
        postpone_ping();
    }
    void do_ping() {
        std::ostream out(&write_buf_); out << "ping\n";
        service.post( MEM_FN2(step,error_code(),0));
    }
};
```

```

    void postpone_ping() {
        timer_.expires_from_now(boost::posix_time::millisec(rand() %
7000));
        timer_.async_wait( MEM_FN(do_ping));
    }
    void do_ask_clients() {
        std::ostream out(&write_buf_); out << "ask_clients\n";
    }
};

```

The example is a bit more complex, since we need to ping the server at random times. To do this, we postpone a ping operation after we successfully ask for the list of clients for the first time. Then, on each ping answer from the server, we postpone another ping operation.

To run it all, use the following code snippet:

```

int main(int argc, char* argv[]) {
    ip::tcp::endpoint ep( ip::address::from_string("127.0.0.1"),
8001);
    talk_to_svr::start(ep, "John");
    service.run();
}

```

Going with co-routines, we saved 15 lines of code, and the code became much easier to read.

We've barely touched the subject of co-routines here. If you want more information, please check out the author's web page, http://blog.think-async.com/2010_03_01_archive.html.

Summary

We've seen how Boost.Asio plays nice with STL streams and `streambuf` objects. We've seen how co-routines make our code more compact and easier to read.

Time to bring the big guns, such as Asio versus Boost.Asio, advanced debugging, SSL, and some platform-dependent features.

7

Boost.Asio – Advanced Topics

This chapter deals with some of the advanced topics of Boost.Asio. It's unlikely that you'll need to delve into these for day-to-day programming, but they are definitely good to know:

- If debugging fails, you'll see what Boost.Asio can do to help
- If you need to deal with SSL, see how much Boost.Asio can help you
- If you target a specific OS, see what extra features Boost.Asio has in store for you

Asio versus Boost.Asio

The author of Boost.Asio also maintains Asio. You can think of it as Asio, as it comes in two flavours, Asio (non-Boost) and Boost.Asio. The author claims that the updates will always appear in non-Boost Asio first, and periodically, they will be incorporated into the Boost distribution.

The differences in a nutshell are as follows:

- Asio definitions are in namespace `asio::` while Boost.Asio definitions are in `boost::asio::`
- Main header in Asio is `asio.hpp` and `boost/asio.hpp` for Boost.Asio
- Asio has a class for launching threads (the equivalent for `boost::thread`)
- Asio provides its own error code classes (`asio::error_code` instead of `boost::system::error_code` and `asio::system_error` instead of `boost::system::system_error`)

You can find more details about Asio at, <http://think-async.com>.

You should decide for yourself which version you prefer; personally, I prefer Boost.Asio. Here are a few things to consider when making your choice:

- New versions of Asio are released faster than new versions of Boost.Asio (since new versions of Boost occur quite seldomly)
- Asio is header-only (while some parts of Boost.Asio depend on other Boost libraries, which might need to be compiled)
- Both Asio and Boost.Asio are quite mature, so unless you're longing for some Asio features about to be released, Boost.Asio is quite a safe bet, and you'll have the other Boost libraries at your disposal as well

You can use Asio and Boost.Asio in the same application, although, I would not recommend it. This can happen silently, in which case it's ok, for instance, if you're using Asio, and some third party library is using Boost.Asio or vice versa.

Debugging

Debugging synchronous applications is usually easier than debugging asynchronous applications. For a synchronous application, if it gets blocked, you'll just break into debug, and you'll get a picture of where you are (synchronous means sequential). However, when going asynchronous, events don't happen sequentially, so in case of a bug, it's really hard to track what happened.

To avoid this, first, you should delve deeply into co-routines. If implemented correctly, you should virtually have no problems at all.

Just in case you do, when it comes to asynchronous programming, Boost.Asio lends you a helping hand; Boost.Asio allows for **handler tracking**, when the macro `BOOST_ASIO_ENABLE_HANDLER_TRACKING` is defined. If so, Boost.Asio writes a lot of aiding output to the standard error stream, recording the time, the asynchronous operation, and the relationship to its completion handler.

Handler tracking information

The information is not that easy to grasp, but it's very useful nonetheless. The Boost.Asio's output is `@asio|<timestamp>|<action>|<description>`.

The first tag is always `@asio`, and you can use that to easily filter the messages coming from Boost.Asio, in case other sources dump to the standard error stream (the equivalent of `std::cerr`) as well. The `timestamp` instance is in seconds and microseconds since 1st January, 1970 UTC. The `action` instance can be any of the following:

- `>n`: This is used when we enter the handler number `n`. The `description` instance contains the arguments that were sent to the handler.
- `<n`: This is used when we exit the handler number `n`.
- `!n`: This is used when we exited the handler number `n` due to an exception.
- `~n`: This is used when the handler with number `n` is destroyed without being called; probably, because the `io_service` instance is destroyed too soon (before `n` gets a chance to be called).
- `n*m`: This is used when the handler number `n` creates a new asynchronous operation with completion handler with number `m`. This is the start of the asynchronous operation, as the `description` instance shows. The completion handler is called when you see `>m` (start) and `<m` (end).
- `n`: This is used when the handler number `n` performs an operation, as shown in `description` (which can be a `close` or a `cancel` operation). You usually can safely ignore these messages.

Whenever `n` is 0, the operation is performed outside of any (asynchronous) handler; you'll usually see this as the first operation(s), or in case you're working with signals, whenever a signal is triggered.

You should pay close attention to messages of type `!n` and `~n`, which are most likely errors in your code. In the former case, the asynchronous functions don't throw exceptions, thus, the exception must have been generated by you; you should not allow exceptions to exit your completion handlers. In the latter case, you probably destroyed the `io_service` instance too soon, before all completion handlers were called.

An example

To show you an example of this aiding output, let's modify the example used in *Chapter 6, Boost.Asio Other Features*. All you need to do is add an extra `#define` before including `boost/asio.hpp`:

```
#define BOOST_ASIO_ENABLE_HANDLER_TRACKING
#include <boost/asio.hpp>
...
```

Also, we're dumping to console when the user logs in and when he receives the first list of clients. The output will be:

```
@asio|1355603116.602867|0*1|socket@008D4EF8.async_connect
@asio|1355603116.604867|>1|ec=system:0
@asio|1355603116.604867|1*2|socket@008D4EF8.async_send
@asio|1355603116.604867|<1|
@asio|1355603116.604867|>2|ec=system:0,bytes_transferred=11
@asio|1355603116.604867|2*3|socket@008D4EF8.async_receive
@asio|1355603116.604867|<2|
@asio|1355603116.605867|>3|ec=system:0,bytes_transferred=9
@asio|1355603116.605867|3*4|io_service@008D4BC8.post
@asio|1355603116.605867|<3|
@asio|1355603116.605867|>4|
John logged in
@asio|1355603116.606867|4*5|io_service@008D4BC8.post
@asio|1355603116.606867|<4|
@asio|1355603116.606867|>5|
@asio|1355603116.606867|5*6|socket@008D4EF8.async_send
@asio|1355603116.606867|<5|
@asio|1355603116.606867|>6|ec=system:0,bytes_transferred=12
@asio|1355603116.606867|6*7|socket@008D4EF8.async_receive
@asio|1355603116.606867|<6|
@asio|1355603116.606867|>7|ec=system:0,bytes_transferred=14
@asio|1355603116.606867|7*8|io_service@008D4BC8.post
@asio|1355603116.607867|<7|
@asio|1355603116.607867|>8|
John, new client list: John
```

Lets analyse this line by line:

- We enter `async_connect`, which creates handler 1 (in our case, all handlers are `talk_to_svr::step`)
- Handler 1 is called (on successful connection to server)
- Handler 1 calls `async_send`, which creates handler 2 (here, we're sending the login message to the server)
- Handler 1 exits
- Handler 2 is called, and 11 bytes are sent (`login John`)
- Handler 2 calls `async_receive`, which creates handler 3 (we're waiting for server to answer to our login)

- Handler 2 exits
- Handler 3 is called, and we received 9 bytes (`login ok`)
- Handler 3 posts `on_answer_from_server` (which creates handler 4)
- Handler 3 exits
- Handler 4 is called, which then dumps `John logged in`
- Handler 4 posts another step (handler 5), which will write `ask_clients`
- Handler 4 exits
- Handler 5 enters
- Handler 5, `async_send ask_clients`, creates handler 6
- Handler 5 exits
- Handler 6 enters (we successfully send `ask_clients` to server)
- Handler 6 calls `async_receive`, which creates handler 7 (we're waiting for the server to send us the list of existing clients)
- Handler 6 exits
- Handler 7 is called, and we received the client list
- Handler 7 posts `on_answer_from_server` (which creates handler 8)
- Handler 7 exits
- Handler 8 enters, and dumps the client list (`on_clients`)

It will take a while to get used to, but once you grasp it, you'll isolate the output that contains the problem, and find the actual piece of code that needs to be fixed.

Handler tracking to file

By default, the handler tracking information is dumped to the standard error stream (the equivalent of `std::cerr`). It's very likely that you'll want to redirect this output somewhere else. For one thing, by default, for a console application, the output and error output dump to the same place, that is, the console. But for a Windows (non-console) application, the default error stream is null.

You can redirect the error output via the command line, such as:

```
some_application 2>err.txt
```

Or, if you're not too lazy, you can do it programmatically, as given in the following code snippet:

```
// for Windows
HANDLE h = CreateFile("err.txt", GENERIC_WRITE, 0, 0, CREATE_ALWAYS,
FILE_ATTRIBUTE_NORMAL, 0);
SetStdHandle(STD_ERROR_HANDLE, h);
// for Unix
int err_file = open("err.txt", O_WRONLY);
dup2(err_file, STDERR_FILENO);
```

SSL

Boost.Asio provides classes for some basic SSL support. Behind the scenes, it uses OpenSSL. So, if you want to use SSL, first download OpenSSL from www.openssl.org and build it. You should note that, usually, building OpenSSL is no easy task, especially if you don't have a popular compiler, such as Visual Studio.

Assuming you have OpenSSL built successfully, Boost.Asio has some wrapper classes around it:

- `ssl::stream`: It guides you to what to use instead of the `ip::<protocol>::socket` class
- `ssl::context`: This is the context for initial handshake
- `ssl::rfc2818_verification`: This class is the easy way to verify a certificate against a hostname according to the rules from RFC 2818

First, you create and initialize the SSL context, then open a socket using the given context and the given remote host, connect to the remote host, and do the SSL handshake. Once the handshake is over, you can use the Boost.Asio's `read*/write*` free functions.

Here's a simple example of an HTTPS client that connects to Yahoo!:

```
#include <boost/asio.hpp>
#include <boost/asio/ssl.hpp>
using namespace boost::asio;
io_service service;
int main(int argc, char* argv[]) {
```

```

typedef ssl::stream<ip::tcp::socket> ssl_socket;
ssl::context ctx(ssl::context::sslv23);
ctx.set_default_verify_paths();
// Open an SSL socket to the given host
io_service service;
ssl_socket sock(service, ctx);
ip::tcp::resolver resolver(service);
std::string host = "www.yahoo.com";
ip::tcp::resolver::query query(host, "https");
connect(sock.lowest_layer(), resolver.resolve(query));
// The SSL handshake
sock.set_verify_mode(ssl::verify_none);
sock.set_verify_callback(ssl::rfc2818_verification(host));
sock.handshake(ssl_socket::client);
std::string req = "GET /index.html HTTP/1.0\r\nHost: "
    + host + "\r\nAccept: */*\r\nConnection: close\r\n\r\n";
write(sock, buffer(req.c_str(), req.length()));
char buff[512];
boost::system::error_code ec;
while ( !ec ) {
    int bytes = read(sock, buffer(buff), ec);
    std::cout << std::string(buff, bytes);
}
}

```

The first lines are pretty self-explanatory. When you connect to the remote host, you use `sock.lowest_layer()`, in other words, you use the underlying socket (since `ssl::stream` is just a wrapper). The next three lines perform the handshake. Once that is done, you make the HTTP request with the Boost.Asio's `write()` function, and read (`read()`) all incoming bytes.

When implementing SSL servers, things get a bit more complicated. Boost.Asio comes with an example of an SSL server, which you'll find in `boost/libs/asio/example/ssl/server.cpp`.

Boost.Asio Windows features

The features that follow apply only to the Windows operating system.

Stream Handles

Boost.Asio allows you to create a wrapper over a Windows Handle, after which you can use most of the free functions, such as `read()`, `read_until()`, `write()`, `async_read()`, `async_read_until()`, and `async_write()`. Here's how to read a line from a file:

```
HANDLE file = ::CreateFile("readme.txt", GENERIC_READ, 0, 0,
    OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL | FILE_FLAG_OVERLAPPED, 0);
windows::stream_handle h(service, file);
streambuf buf;
int bytes = read_until(h, buf, '\n');
std::istream in(&buf);
std::string line;
std::getline(in, line);
std::cout << line << std::endl;
```

The `stream_handle` class is available only when the I/O completion port backend is used (which is the default). If this is the case, `BOOST_ASIO_HAS_WINDOWS_STREAM_HANDLE` is defined.

Random access Handles

Boost.Asio allows you to do random-access read and write operations on Handles that refer to regular files. Again, you create a wrapper over a Handle, and then use the free functions, such as `read_at()`, `write_at()`, `async_read_at()`, or `async_write_at()`. To read 50 characters starting at offset 1,000, you'll use the following code snippet:

```
HANDLE file = ::CreateFile("readme.txt", GENERIC_READ, 0, 0,
    OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL | FILE_FLAG_OVERLAPPED, 0);
windows::random_access_handle h(service, file);
char buf[50];
int bytes = read_at(h, 1000, buffer(buf));
std::string msg(buf, bytes);
std::cout << msg << std::endl;
```

For Boost.Asio, the random access Handlers only provide random access, you cannot use them as Stream Handles. In other words, the free functions, such as `read()`, `read_until()`, `write()`, and their asynchronous counterparts cannot be used with a random access Handle.

The `random_access_handle` class is available only when the I/O completion port backend is used (which is the default). If this is the case, `BOOST_ASIO_HAS_WINDOWS_RANDOM_ACCESS_HANDLE` is defined.

Object Handles

You can wait on Windows Handles to kernel objects, such as change notification, console input, event, memory resource notification, process, semaphore, thread, or waitable timer. Or simply put, anything you can call `WaitForSingleObject` on. For them, you can create a `object_handle` wrapper, and use `wait()` or `async_wait()` on it:

```
void on_wait_complete(boost::system::error_code err) {}
...
HANDLE evt = ::CreateEvent(0, true, true, 0);
windows::object_handle h(service, evt);
// synchronous wait
h.wait();
// asynchronous wait
h.async_wait(on_wait_complete);
```

Boost.Asio POSIX features

The features that follow apply only on Unix operating systems.

Local sockets

Boost.Asio provides basic support for local sockets (also known as Unix domain sockets).

A local socket is a socket that can only be accessed from the applications that run on the host machine. You can use local sockets for easy inter-process communication. You can connect both as a client socket or as a server socket. For local sockets, the endpoint is a filename, such as `/tmp/whatever`. One cool thing is that you can assign rights to that given file, and therefore, disallow certain users on your machine from creating a socket to the file.

You can connect as a client socket, as given in the following code snippet:

```
local::stream_protocol::endpoint ep("/tmp/my_cool_app");
local::stream_protocol::socket sock(service);
sock.connect(ep);
```

You can create a server socket, as given in the following code snippet:

```
::unlink("/tmp/my_cool_app");
local::stream_protocol::endpoint ep("/tmp/my_cool_app");
local::stream_protocol::acceptor acceptor(service, ep);
local::stream_protocol::socket sock(service);
acceptor.accept(sock);
```

Once the socket is successfully created, you can use it just like a regular socket; it has the same member functions as the other socket classes, and you can use the free functions that use sockets as well.

Note that local sockets are available only if the target operating system supports them, namely, `BOOST_ASIO_HAS_LOCAL_SOCKETS` (if it is defined).

Connecting local sockets

Finally, you can connect two sockets, either connectionless (datagram), or connection oriented (streams):

```
// connection oriented
local::stream_protocol::socket s1(service);
local::stream_protocol::socket s2(service);
local::connect_pair(s1, s2);
// connection-less
local::datagram_protocol::socket s1(service);
local::datagram_protocol::socket s2(service);
local::connect_pair(s1, s2);
```

Internally, `connect_pair` uses the infamous POSIX `socketpair()` function. What this basically does is connect two sockets without the complicated socket creation process; just one line of code, and you're done. This used to be an easy way for inter-thread communication. While in modern programming, you can avoid it, and you could find it useful when dealing with legacy code that uses sockets.

POSIX file descriptors

Boost.Asio allows synchronous and asynchronous operations on some POSIX file descriptors, such as pipes, standard I/O, and other devices (but not on regular files).

Once you create a `stream_descriptor` instance for such a POSIX file descriptor, you can use some of the free functions provided by Boost.Asio, such as `read()`, `read_until()`, `write()`, `async_read()`, `async_read_until()`, and `async_write()`.

Here's how you read one line from `stdin` and dump it to `stdout`:

```
size_t read_up_to_enter(error_code err, size_t bytes) { ... }
posix::stream_descriptor in(service, ::dup(STDIN_FILENO));
posix::stream_descriptor out(service, ::dup(STDOUT_FILENO));
char buff[512];
int bytes = read(in, buffer(buff), read_up_to_enter);
write(out, buffer(buff, bytes));
```

The `stream_descriptor` class is available only if the target operating system supports it, namely, `BOOST_ASIO_HAS_POSIX_STREAM_DESCRIPTOR` (if it is defined).

Fork

Boost.Asio supports programs that make use of the `fork()` system call. You need to tell the `io_service` instance when the `fork()` function is about to happen and when it has happened. Refer to the following code snippet:

```
service.notify_fork(io_service::fork_prepare);
if (fork() == 0) {
    // child
    service.notify_fork(io_service::fork_child);
    ...
} else {
    // parent
    service.notify_fork(io_service::fork_parent);
    ...
}
```

This advises to use `service` that it's about to be called on a different thread. Even though Boost.Asio allows for this, I strongly recommend you use threads, as using `boost::thread` is a piece of cake.

Summary

Strive for your code to be simple and easy to understand. Learn and use co-routines. This will minimize the debugging you need to do, but just in case there are still some bugs lurking in the code, Boost.Asio lends a helping hand, as we've seen in the *Debugging* section.

In case you need to deal with SSL, Boost.Asio allows for basic SSL programming.

Finally, if you know your application is targeted at a given OS, you can take advantage of the features Boost.Asio provides for that specific operating system.

Network programming is crucial nowadays. Boost.Asio is a must for any C++ programmer of the 21st century. We've also delved into theory, then into practice; use this as both a reference and a hands-on collection of Boost.Asio examples, as you can easily read, test, understand, and extend. Hope it's been a fun to read. And it's definitely been a fun to write!

Index

Symbols

***_at functions**

- about 39
- async_read_at() 39
- async_write_at() 39
- read_at() 39
- write_at() 39

A

Asio

- about 127
- versus Boost.Asio 127, 128

assign(protocol,socket) function 25

async_call function 104

async_connect(endpoint) function 25

**async_connect(socket, begin [, end]
[, condition],handler function** 35

asynchronous client 78-81

asynchronous I/O

- client applications 96, 97
- server applications 98-101

asynchronous operations 104-107

asynchronous programming

- about 8, 9, 11 40
- asynchronous work 47, 48
- mixing, with synchronous programming 87, 88
- need for 40-44
- poll() 44
- poll_one() 44, 46
- run() 44
- run() function 44, 45
- run_one() 44
- run_one() function 45

- service.dispatch(handler) 50

- service.post(handler) 50

- service.wrap(handler) 50

asynchronous server 82-86

- threading 101-103

async_op function 105

async_read_at() function 39

**async_read_until(stream, stream_buffer,
completion, handler) function** 38

**async_read_until(stream, stream_buffer,
delim, handler function** 38

async_write_at() function 39

async_write function 122

at_mark() function 30

B

Berkeley Software Distribution. *See* BSD

bind(endpoint) function 25

Boost.Asio

- about 5

- Boost::asio::buffer() 113

- Boost::asio::streambuf 113

- building 7

- co-routines 120-125

- features 113

- input/output facilities 13

- io_service class 15-18

- macros 8

- POSIX features 135

- STL streams 114, 115

- threading 12

- timers 14, 15

- versus Asio 127, 128

- Windows features 133

- wrapper classes 132

Boost.Asio namespaces

- boost::asio 21
- Boost::asio::error 21
- Boost::asio::ip 21
- Boost::asio::local 21
- Boost::asio::ssl 21
- Boost::asio::windows 21

boost::bind functor 53

Boost.DateTime 7

Boost.Regex 7

Boost.System 7

Borland C++ 5.9.2+ 6

BSD 6

buffer() function 34

buffer function wrapper 33

C

cancel() function 25

client

- passing, to server messages 88, 89
- requests 69

client applications

- asynchronous I/O 96, 97
- synchronous I/O 89, 90

client list changed event 83

close() function 25

commit(n) function 116

completion function 106

connect(endpoint) function 25

connect functions 35

connection::ptr(new connection)->start(ep) 54

connect(socket, begin [, end] [, condition] 35

consume(n) function 116

co-routines 120-125

D

data() function 116

deadline_timer timer_ function 80

debugging

- about 128
- example 129-131
- handler tracking 128, 129
- handler tracking, to file 131

dispatch() 51

do_complete() function 65

do_read() function 63, 65

do_write() 55

do_write() function 63, 65

E

Echo client 57, 58

Echo server 58

endpoints 22

error codes 11, 12

exceptions 11, 12

F

fork() function 137

G

get_io_service() function 28

get_option(option) function 28

H

handle_clients() function 41

handler tracking 128

I

IMCP 6

Instant Private Network. *See* IPN

Internet Control Message Protocol. *See* IMCP

io_control(cmd) function 28

io_service class 15

io_servicestrand** class 52

ip::address

- from_string(str) function 22

ip::addressto_string()** function 22

ip::address_v4

- any() function 22

- broadcast([addr, mask]) function 22

ip::address(v4_or_v6_address) function 22

ip::host_name() function 22

IPN 5

is_open() function 25

L

libtorrent 5

local_endpoint() function 30

local sockets

about 135

connecting 136

M

max_size()function 116

N

native_handle() function 30

native_non_blocking() function 30

Network API

about 21

Boost.Asio namespaces 21

connect functions 35

endpoints 22, 23

IP addresses 22

read/write functions 36

sockets 23, 24

non_blocking() function 30

non-Boost. *See* Asio

O

on_new_client() function 93

on_read_msg() function 42

open(protocol) function 25

OpenSSL 7

out-of-band (OOB) 27

P

PokerTH 5

poll() function 47

poll_one function 46

POSIX features, Boost.Asio

file descriptors 136

Fork 137

local sockets 135

local sockets, connecting 136

prepare(n) function 116

proxies

asynchronous proxy example 109-111

implementing 108

R

read_at() functions 39

read operation 111

read_request() function 76

read_until(stream, stream_buffer, completion) function 38

read_until(stream, stream_buffer, delim) function 38

read/write functions 25

about 36

async_read() 36

async_read_until() functions 38

async_write() 36

***_at** functions 39, 40

end conditions 36-38

read() 36

write() 36

reenter(entry) class 122

Remobo 5

remote_endpoint() function 30

run() function 106

run_one() function 45

S

server applications

asynchronous I/O 98-101

synchronous I/O 92, 93

server messages

passing, to client 88, 89

service.dispatch(handler) 50

service.post(handler) 50

service.run() function 97

service.run() loop 10

service.wrap(handler) handler 50

set_option(option) function 28

set_reading() function 43

shared_buffer class 33, 34

shared_ptr_from_this() function 62

shutdown(type_of_shutdown) function 25

size()function 116

socket member functions

- Read/write functions 25-28
- related functions, connecting 25
- socket control 28, 29
- TCP versus UDP 30
- UDP versus ICMP 30

socketpair() function 136

sockets

- about 23, 24
- buffer function wrapper 33, 34
- considerations 31
- member functions 24
- socket buffers 31, 32
- synchronous error codes 24

some_function 48

SSL 132, 133

STL streams

- Boost.Asio 114, 115

streambuf class 116

streambuf() function 116

streambuf objects

- async_read_at() function 119
- async_read() function 119
- async_read_until() function 119
- async_write_at() function 119
- async_write() function 119
- read_at() function 118
- read() function 118
- read_until() function 118
- write_at() function 119
- write() function 118

synchronous client 70-72

synchronous I/O

- client applications 89, 90
- server applications 92, 93

synchronous programming

- about 8, 9
- mixing, with asynchronous programming 87, 88

synchronous server

- about 73-77
- threading 94-96

T

TCP 6

TCP asynchronous client 61-64

TCP asynchronous server 64-66

TCP Echo server/clients

- about 58
- asynchronous client 61-64
- asynchronous server 64-66
- code 66
- synchronous client 59, 60
- synchronous server 60, 61

TCP synchronous client 59, 60

TCP synchronous server 60, 61

threading

- asynchronous server 101-103
- in synchronous server 94-96

threading, Boost.Asio

- io_service class 12
- socket class 13
- utility class 13

timers 14

Transmission Control Protocol. *See* TCP

typedef keyword 24

U

UDP 6

UDP Echo server/clients

- about 66
- synchronous Echo client 67
- synchronous Echo server 68

UDP synchronous Echo client 67

UDP synchronous Echo server 68

update_clients_changed() function 103

User Datagram Protocol. *See* UDP

W

Windows features, Boost.Asio

- object handles 135
- random access handles 134
- stream handles 134

write_at() function 39

Y

yield code class 122



Thank you for buying **Boost.Asio C++ Network Programming**

About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: www.packtpub.com.

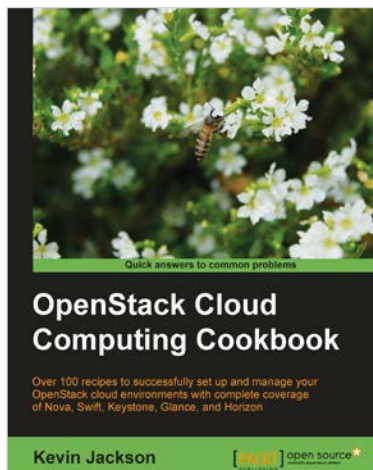
About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licences, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.



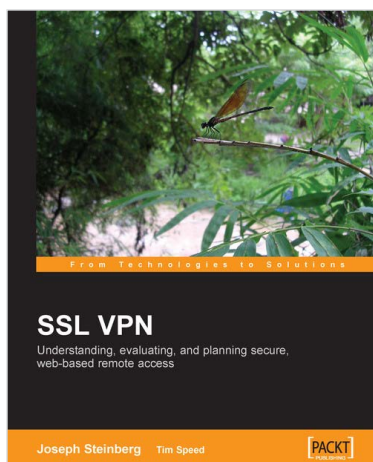
OpenStack Cloud Computing Cookbook

ISBN: 978-1-849517-32-4

Paperback: 318 pages

Over 100 recipes to successfully set up and manage your OpenStack cloud environment with complete coverage of Nova, Swift, Keystone, Glance, and Horizon

1. Learn how to install and configure all the core components of OpenStack to run an environment that can be managed and operated just like AWS or Rackspace
2. Master the complete private cloud stack from scaling out compute resources to managing swift services for highly redundant, highly available storage



SSL VPN

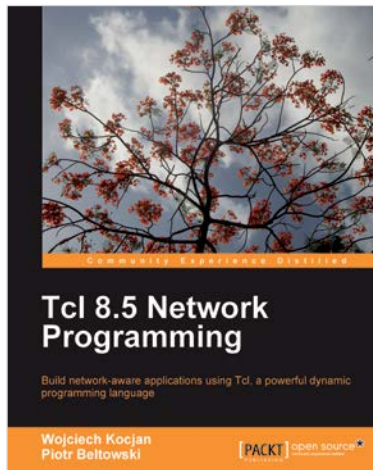
ISBN: 978-1-904811-07-7

Paperback: 212 pages

Understanding, evaluating, and planning secure, web-based remote access

1. Understand how SSL VPN technology works
2. Evaluate how SSL VPN could fit into your organisation's security strategy
3. Practical advice on educating users, integrating legacy systems, and eliminating security loopholes

Please check **www.PacktPub.com** for information on our titles

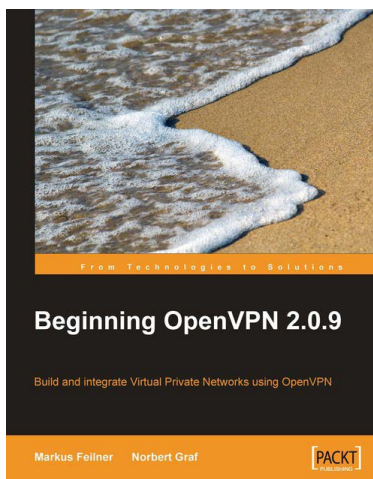


Tcl 8.5 Network Programming

ISBN: 978-1-849510-96-7 Paperback: 588 pages

Build network-aware applications using Tcl, a powerful dynamic programming language

1. Develop network-aware applications with Tcl
2. Implement the most important network protocols in Tcl
3. Packed with hands-on-examples, case studies, and clear explanations for better understanding



Beginning OpenVPN 2.0.9

ISBN: 978-1-847197-06-1 Paperback: 356 pages

Build and integrate Virtual Private Networks using OpenVPN

1. A practical guide to using OpenVPN for building both basic and complex Virtual Private Networks (VPNs)
2. Learn how to make use of OpenVPNs modules, high-end-encryption and how to combine it with servers for your individual privacy
3. Advanced management of security certificates

Please check **www.PacktPub.com** for information on our titles